



Concurrent Building Blocks (Part 1)



Concurrent and Parallel Programming

Reusable building blocks

- ▶ One of the goals of object oriented programming is to favor the **reusability of pre-existing code**.
- ▶ The Java library contains many **reusable building blocks, which also include specific ones for concurrent programming**.
- ▶ Use of existing building blocks, reduces the amount of work required for development, as well as the associated risks and maintenance costs.

Reusable building blocks

- ▶ In several situations, it is required to **modify and extend** the functionality provided by the existing building blocks, for example by means of inheritance.
- ▶ In a multi-threaded application, it is important that such modifications or extensions are performed **without compromising the thread-safety** of the building block.

Synchronized collections

- ▶ **Synchronized collections** already existed in versions prior to Java 5. Included are:
 - ▶ Vector and HashTable (from the first version of JDK)
 - ▶ All the complements introduced in JDK 1.2 (e.g.: Stack)
- ▶ Synchronized wrapper classes that are created with the *Collections.synchronized...* methods (e.g. for ArrayList)

no longer
used!



Synchronized collections

		Implementations				
		Hash Table	Resizable Array	Balanced Tree	Linked List	Hash Table + Linked List
Interfaces	Set	HashSet		TreeSet		LinkedHashSet
	List		ArrayList		LinkedList	
	Deque		ArrayDeque		LinkedList	
	Map	HashMap		TreeMap		LinkedHashMap

Example of how to create a synchronized wrapper:

```
List synchList = Collections.synchronizedList(new ArrayList());
```

Synchronized collections

The Collections class provides:

<code>static <T> Collection<T></code>	<code>synchronizedCollection(Collection<T> c)</code> Returns a synchronized (thread-safe) collection backed by the specified collection.
<code>static <T> List<T></code>	<code>synchronizedList(List<T> list)</code> Returns a synchronized (thread-safe) list backed by the specified list.
<code>static <K,V> Map<K,V></code>	<code>synchronizedMap(Map<K,V> m)</code> Returns a synchronized (thread-safe) map backed by the specified map.
<code>static <T> Set<T></code>	<code>synchronizedSet(Set<T> s)</code> Returns a synchronized (thread-safe) set backed by the specified set.
<code>static <K,V> SortedMap<K,V></code>	<code>synchronizedSortedMap(SortedMap<K,V> m)</code> Returns a synchronized (thread-safe) sorted map backed by the specified sorted map.
<code>static <T> SortedSet<T></code>	<code>synchronizedSortedSet(SortedSet<T> s)</code> Returns a synchronized (thread-safe) sorted set backed by the specified sorted set.

Disadvantages

- ▶ Synchronized collections support thread-safety by **serializing all accesses**. The wrapper object is used as implicit lock for all of its delegating methods.
- ▶ Consequence of this approach is **poor concurrency!**
 - ▶ For many operations, the wrapper has to keep the lock during the whole time of execution.
 - ▶ Strongly **affected are all the operations that run across the collection**, including all traversals indirectly performed by methods such as `equals()`, `containsAll()`, `hashCode()`, ...

Disadvantages

- ▶ In addition, synchronized collections launch **ConcurrentModificationExceptions** when trying to modify the collection while looping through it.
- ▶ Iterators and collection loops (for-each loop) are therefore considered **fail-fast**.

Example


```
public class CompoundAction {  
    public static void main(final String[] args) {  
        final List<Integer> myList = Collections  
            .synchronizedList(new ArrayList<Integer>());  
        myList.add(100);  
        myList.add(200);  
        myList.add(300);  
        myList.add(400);  
        myList.add(500);  
  
        for (final Integer integer : myList) {  
            System.out.println(integer);  
            myList.remove(integer); ← Not allowed  
        }  
    }  
}
```

Exception in thread "main"
java.util.ConcurrentModificationException
at java.util.ArrayList\$Itr.checkForComodification(Unknown Source)
at java.util.ArrayList\$Itr.next(Unknown Source)
at examples.CompoundAction.main(CompoundAction.java:17)

Example

```
public class CompoundAction {  
    public static void main(final String[] args) {  
        final List<Integer> myList = Collections  
            .synchronizedList(new ArrayList<Integer>());  
  
        myList.add(100);  
        myList.add(200);  
        myList.add(300);  
        myList.add(400);  
        myList.add(500);  
  
        for (final Iterator iterator = myList.iterator(); iterator.hasNext();) {  
            final Integer integer = (Integer) iterator.next();  
            System.out.println(integer);  
            iterator.remove();  
        }  
    }  
}
```

Solution: use the iterator



But in a multi-threaded program the
problem might be hidden, ...

Example

```

public class MTCompoundAction {
    static volatile boolean isRunning = true;
    static List<Integer> list = Collections.
        synchronizedList(new ArrayList<Integer>());

    public static void main(final String[] args) {
        final Thread thread = new Thread(new Filter());
        thread.start();
        for (int i = 1; i <= 1000; i++) {
            list.add(i);
            if ((i % 100) == 0)
                System.out.println("Added " + i + " elements");
        }
        isRunning = false;

        //...
        System.out.println("Size=" + list.size());
    }
}

class Filter implements Runnable {
    public void run() {
        while (MTCompoundAction.isRunning) {
            final Iterator<Integer> it =
                MTCompoundAction.list.iterator();
            while (it.hasNext()) {
                if (it.next().intValue() % 2 == 0)
                    it.remove();
            }
        }
    }
}

Exception in thread "Thread-0" Added 100 elements
Added 200 elements
Added 300 elements
Added 400 elements
Added 500 elements
Added 600 elements
Added 700 elements
Added 800 elements
java.util.ConcurrentModificationException
at java.util.ArrayList$Itr.checkForComodification(ArrayList.java:901)
at java.util.ArrayList$Itr.next(ArrayList.java:851)
at lesson06.Filter.run(MTCompoundAction.java:15)
at java.lang.Thread.run(Thread.java:745)
Added 900 elements
Added 1000 elements
Size=967

```

Synchronized collections

The Javadoc provides the following information:

- ▶ "In order to guarantee serial access, it is critical that **all access to the backing collection is accomplished through the returned collection**. It is imperative that the user manually synchronize on the returned collection when traversing it (**by using the wrapper as a lock**):

```
Collection syncCollection =  
    Collections.synchronizedCollection(myCollection);  
synchronized (syncCollection) {  
    Iterator i = syncCollection.iterator(); // Must be in the synchronized block  
    while (i.hasNext())  
        performOperation(i.next());  
}
```

Failure to follow this advice may result in **non-deterministic behavior**."

Client-side locking

- ▶ This solution is called **client-side locking** because the intrinsic lock is acquired outside the object.
- ▶ This solution has to be used **for all the compound actions performed by the client code.**
- ▶ The general structure is:

```
synchronized (myList) {  
    Iterator<Integer> it = myList.iterator();  
    while (it.hasNext())  
        doSomething(it.next());  
}
```

Example

```

public class MTCompoundAction {
    static volatile boolean isRunning = true;
    static List<Integer> list = Collections.
        synchronizedList(new ArrayList<Integer>());

    public static void main(final String[] args) {
        final Thread thread = new Thread(new Filter());
        thread.start();

        for (int i = 1; i <= 1000; i++) {
            list.add(i);
            if ((i % 100) == 0)
                System.out.println("Added " + i + " elements");
        }
        isRunning = false;

        try {
            thread.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        System.out.println("List=" + list.size());
    }
}

class Filter implements Runnable {
    public void run() {
        while (MTCompoundAction.isRunning) {

            synchronized (MTCompoundAction.list) {
                final Iterator<Integer> it =
                    MTCompoundAction.list.iterator();

                while (it.hasNext()) {
                    if (it.next().intValue() % 2 == 0)
                        it.remove();
                }
            }
        }
    }
}

```

The whole traversal has to be performed with client-side locking!

Disadvantages

- ▶ Client side-locking can lead to **deadlock and starvation problems**, but most of all, it suffers **of scalability difficulties**:
 - ▶ We know that the longer a lock is maintained by a thread, the more likely it will be contended by other threads.

Disadvantages

- ▶ To summarize, synchronized collections:
 - ▶ are hard to extend. In particular, it is not trivial to add new compound actions.
 - ▶ Can suffer from performance problems. All methods use the same collection-wide lock.

Immutable collections

- ▶ An effective alternative to synchronized collections takes inspiration from the **technique of the immutable holder**, introduced in the last lesson.
- ▶ Java provides specific implementations for **collections that take advantage of immutability: unmodifiable collections**.

Unmodifiable collections

- ▶ Unmodifiable collections are similar to immutable objects (but are not completely immutable).
- ▶ Provide read-only views (wrappers) of collection objects. It is not possible to add or remove items, but the underlying collection can still be modified.

The
Collections
class
provides:

<code>static <T> Collection<T></code>	<code>unmodifiableCollection(Collection<? extends T> c)</code> Returns an unmodifiable view of the specified collection.
<code>static <T> List<T></code>	<code>unmodifiableList(List<? extends T> list)</code> Returns an unmodifiable view of the specified list.
<code>static <K,V> Map<K,V></code>	<code>unmodifiableMap(Map<? extends K,? extends V> m)</code> Returns an unmodifiable view of the specified map.
<code>static <T> Set<T></code>	<code>unmodifiableSet(Set<? extends T> s)</code> Returns an unmodifiable view of the specified set.
<code>static <K,V> SortedMap<K,V></code>	<code>unmodifiableSortedMap(SortedMap<K,? extends V> m)</code> Returns an unmodifiable view of the specified sorted map.
<code>static <T> SortedSet<T></code>	<code>unmodifiableSortedSet(SortedSet<T> s)</code> Returns an unmodifiable view of the specified sorted set.

Unmodifiable collections

- ▶ Unmodifiable collections can be used:
 - ▶ to obtain **an immutable copy** of an existing collection. In this case NO references to the original collection should be kept.
 - ▶ to allow **read-only access** to the collection by sharing only the unmodifiable wrapper and maintaining the original collection confined (e.g. in combination with the one-writer many-readers approach).
- ▶ **Attention:** unmodifiable collections do not automatically guarantee correct memory visibility. If modified, the read-only views should be **safely republished, or outdated values have to be tolerated.**

Immutable collections

- ▶ To ease the use of the immutable variants of unmodifiable collections, Java 9 provides **specific factory methods**:

```
List<String> list = List.of("One", "Two", "Three", "Four");  
Map<String, String> map = Map.of("One", "1", "Two", "2", "Three", "3");  
Map<String, String> map = Map.ofEntries(  
    new AbstractMap.SimpleEntry<>("One", "1"),  
    new AbstractMap.SimpleEntry<>("Two", "2"),  
    new AbstractMap.SimpleEntry<>("Three", "3"));
```

- ▶ These immutable collections do not accept the null value.
- ▶ When created with identical values, the same objects might be internally referenced, but this behavior is not guaranteed.

Example

```
class SetWorker implements Runnable {  
    public void run() {  
        ImmutableSetExample.sharedSet = Set.of("One", "Two", "Three", "Four");  
    }  
}
```

```
public class ImmutableSetExample {  
    static volatile Set<String> sharedSet = null;  
  
    public static void main(String[] args) {  
        Thread t = new Thread(new SetWorker());  
        t.start();  
  
        while (sharedSet == null) { /* busy wait */ }  
  
        sharedSet.forEach(System.out::print);  
  
        try {  
            t.join();  
        } catch (InterruptedException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

Unmodifiable collections

- ▶ If you try to modify an unmodifiable/immutable collection, `UnsupportedOperationException`s are thrown.
- ▶ It is possible to transfer the contents of a mutable collection to an unmodifiable/immutable version with:

```
List<String> givenList = Arrays.asList("a", "b", "c");  
  
List<String> result = givenList.stream()  
    .collect(Collectors.collectingAndThen(  
        Collectors.toList(), Collections::unmodifiableList));
```

Concurrent building blocks

- ▶ From Java 5, the Java libraries contain a large set of **concurrent building blocks**, including **all concurrent thread-safe collections and all synchronizers** (used to coordinate the cooperation of threads - we'll talk about them in the next lessons).
- ▶ Replacing a synchronized collection with a concurrent collection provides potential improvements in terms of application performances.



Concurrent collections

- ▶ Concurrent collections are designed to support concurrent access by **multiple threads**.
 - ▶ *ConcurrentHashMap* - replaces the synchronized HashMaps
 - ▶ *ConcurrentSkipListMap* - replaces the synchronized TreeMap and other synchronized SortedMaps
 - ▶ *CopyOnWriteArrayList* - replacement for the various versions of synchronized Lists, for situations where traversing (read) is the predominant operation
 - ▶ *CopyOnWriteArraySet* - replacement for the various versions of synchronized Set, for situations where traversing (read) is the predominant operation

Concurrent collections

- ▶ There are also:
 - ▶ The `ConcurrentMap` interface with specific support for compound actions (e.g. conditional remove)
 - ▶ 2 new types of collections: `Queue` and `BlockingQueue` (we will discuss about them in the next lessons)
- ▶ Iterators are **weakly consistent** instead of **fail-fast**:
 - ▶ Concurrent modifications are tolerated (no throw of exceptions)
 - ▶ Iterate on the elements present at construction time of the iterator
 - ▶ May reflect (but it is not guaranteed) modifications on the collection, even after the iterator has been constructed

ConcurrentHashMap

- ▶ **ConcurrentHashMap** is a good substitute of HashMap, when concurrent accesses are required.
- ▶ Optimistic locking is used and, in addition, synchronize methods are replaced with a **fine-grain locking** mechanism called lock-stripping (16 locks by default).
- ▶ `get()` practically never locks, ... only if it is not able to find the element (or if the element reference is null), to verify if the element is actually not present or if it is about to be created.

Example

```
class MapRunner implements Runnable {
    private final String id;
    static Map<String, Integer> map = new HashMap<String, Integer>();

    MapRunner(final String id) {
        this.id = id;
    }

    public void run() {
        final Iterator<String> it = map.keySet().iterator();
        int count = 8;
        while (it.hasNext()) {
            final String key = it.next();
            map.put("Copied" + key, count);
            count++;
        }
        System.out.println("Thread " + id + " completed.");
    }
}

public class HashMapExample {
    public static void main(final String[] args) throws Exception {
        MapRunner.map.put("One", 1);
        MapRunner.map.put("Two", 2);
        MapRunner.map.put("Three", 3);
        MapRunner.map.put("Four", 4);
        MapRunner.map.put("Five", 5);
        new Thread(new MapRunner("A")).start();
        new Thread(new MapRunner("B")).start();
    }
}
```

```
Exception in thread "Thread-1" Exception in thread "Thread-0"  
java.util.ConcurrentModificationException  
at java.util.HashMap$HashIterator.nextEntry(Unknown Source)  
at java.util.HashMap$KeyIterator.next(Unknown Source)  
at MapRunner.run(ConcurrentHashMapExample.java:22)  
at java.lang.Thread.run(Unknown Source)  
java.util.ConcurrentModificationException  
at java.util.HashMap$HashIterator.nextEntry(Unknown Source)  
at java.util.HashMap$KeyIterator.next(Unknown Source)  
at MapRunner.run(ConcurrentHashMapExample.java:22)  
at java.lang.Thread.run(Unknown Source)
```

Example

```
class MapRunner implements Runnable {
    private final String id;
    static Map<String, Integer> map = new ConcurrentHashMap<String, Integer>();

    MapRunner(final String id) {
        this.id = id;
    }

    public void run() {
        final Iterator<String> it = map.keySet().iterator();
        int count = 8;
        while (it.hasNext()) {
            final String key = it.next();
            map.put("Copied" + key, count);
            count++;
        }
        System.out.println("Thread " + id + " completed.");
    }
}

public class ConcurrentHashMapExample {
    public static void main(final String[] args) throws Exception {
        MapRunner.map.put("One", 1);
        MapRunner.map.put("Two", 2);
        MapRunner.map.put("Three", 3);
        MapRunner.map.put("Four", 4);
        MapRunner.map.put("Five", 5);
        new Thread(new MapRunner("A")).start();
        new Thread(new MapRunner("B")).start();
    }
}
```

ConcurrentSkipListMap

- ▶ **ConcurrentSkipListMap** is similar to **ConcurrentHashMap** but **keeps items sorted in the natural order of the keys** (or any other order specified by the user).
- ▶ In general, get/put/contains operations are slower than in **ConcurrentHashMap**, but support for the **SortedMap** and **NavigableMap** interfaces is provided.

ConcurrentMap

- ▶ The ConcurrentMap interface provides several **atomic operations** (similarly to atomic variables), that **can be used to develop thread-safe code**.
- ▶ **Important:** some of these operations can be exploited with **the CAS synchronization idiom (optimistic locking)**, to develop atomic compound actions.
- ▶ In addition, **starting from Java 8**, additional atomic operations that accept **lambda expressions**, have been introduced.

ConcurrentMap

Modifier and Type	Method and Description
default V	compute (K key, BiFunction<? super K,? super V,? extends V> remappingFunction) Attempts to compute a mapping for the specified key and its current mapped value (or null if there is no current mapping).
default V	computeIfAbsent (K key, Function<? super K,? extends V> mappingFunction) If the specified key is not already associated with a value (or is mapped to null), attempts to compute its value using the given mapping function and enters it into this map unless null.
default V	computeIfPresent (K key, BiFunction<? super K,? super V,? extends V> remappingFunction) If the value for the specified key is present and non-null, attempts to compute a new mapping given the key and its current mapped value.
default void	forEach (BiConsumer<? super K,? super V> action) Performs the given action for each entry in this map until all entries have been processed or the action throws an exception.
default V	getOrDefault (Object key, V defaultValue) Returns the value to which the specified key is mapped, or defaultValue if this map contains no mapping for the key.

ConcurrentMap

default V	merge (K key, V value, BiFunction<? super V,? super V,? extends V> remappingFunction) If the specified key is not already associated with a value or is associated with null, associates it with the given non-null value.
V	putIfAbsent (K key, V value) If the specified key is not already associated with a value, associate it with the given value.
boolean	remove (Object key, Object value) Removes the entry for a key only if currently mapped to a given value.
V	replace (K key, V value) Replaces the entry for a key only if currently mapped to some value.
boolean	replace (K key, V oldValue, V newValue) Replaces the entry for a key only if currently mapped to a given value.
default void	replaceAll (BiFunction<? super K,? super V,? extends V> function) Replaces each entry's value with the result of invoking the given function on that entry until all entries have been processed or the function throws an exception.

Can be used with CAS
idiom

Example

```
public static void addNewValue(final String key, final int value) {  
    int previousValue;  
  
    // This code:  
    previousValue = map.putIfAbsent(key, value);  
  
    // ... is identical to this code:  
    if (!map.containsKey(key))  
        previousValue = map.put(key, value);  
    else  
        previousValue = map.get(key);  
  
    // ... But the first one is atomic!!!  
}
```

Optimistic Locking Example (1 / 2)

```
class CASBidder implements Runnable {
    public void run() {
        final String bidkey = "Picasso";

        while (Auction.finished == false) {
            Integer oldBet = null;
            Integer newBet = null;
            do {
                oldBet = Auction.bids.get(bidkey);
                newBet = Integer.sum(oldBet.intValue(), 1);
            } while (!Auction.bids.replace(bidkey, oldBet, newBet));
        }
    }
}

class LambdaBidder implements Runnable {
    public void run() {
        while (Auction.finished == false) {
            Auction.bids.compute("Picasso", (key, oldBet)
                -> Integer.sum(oldBet.intValue(), 1));
        }
    }
}
```

Optimistic Locking Example (2/2)

```
public class Auction {
    static volatile boolean finished = false;
    static final ConcurrentHashMap<String, Integer> bids = new ConcurrentHashMap<>();

    public static void main(String[] args) throws InterruptedException {
        acutionValues.put("Picasso", Integer.valueOf(0));
        acutionValues.put("Monet", Integer.valueOf(0));

        Thread t1 = new Thread(new CASBidder());
        Thread t2 = new Thread(new LambdaBidder());
        t1.start();
        t2.start();
        Thread.sleep(1000);
        finished = true;
        t1.join();
        t2.join();
        bids.forEach((key, value)
            -> System.out.println(key + " -> " + value.intValue()));
    }
}
```

CopyOnWriteArrayList and CopyOnWriteArraySet

- ▶ CopyOnWriteArrayList and CopyOnWriteArraySet are List and Set implementations that allow efficient concurrent access, when frequently iterated (read) and rarely modified.
- ▶ Create and republish a **new copy of the array** for each modification (write).
- ▶ The technique of a **mutable reference to an immutable array** is used (remember: immutable objects do not need synchronization for concurrent access)
- ▶ Also provide atomic operations that accept **lambda expressions**.

```
final CopyOnWriteArrayList<String> threadSafeList = new CopyOnWriteArrayList<String>();
threadSafeList.add("Java");
threadSafeList.add("J2EE");
threadSafeList.add("Collection");

// add, remove not supported by CopyOnWriteArrayList iterator
final Iterator<String> failSafeIterator = threadSafeList.iterator();
while (failSafeIterator.hasNext()) {
    final String s = failSafeIterator.next();
    System.out.printf("Read from CopyOnWriteArrayList : %s %n", s);
    failSafeIterator.remove(); // not supported in CopyOnWriteArrayList
    threadSafeList.remove(s);
}
```

Concurrent collections: weaknesses

- ▶ Concurrent collections are not always a "plug and play" replacement for synchronized collections.
- ▶ The semantic of some methods has been slightly weakened. For example: `size()` and `isEmpty()` return an **estimate instead of an exact value** (in concurrent programs these quantities are "moving targets").
- ▶ **Cannot be locked** for exclusive access (client-side locking is not allowed).
- ▶ The order of the returned object might differ between one iteration and the others.

Concurrent collections and visibility

- ▶ When objects are added or retrieved to/from a concurrent collection **correct memory visibility and extension of the visibility effect** is granted for all states of the objects (in a similar way as for locks and volatile/atomic variables).
- ▶ On the other hand, to grant correct visibility, concurrent collections need also to be **safely published, when shared**.

Summary of topics

- ▶ Introduction to reusable building-blocks
- ▶ Synchronized collections (wrapper classes)
- ▶ Client-side locking
- ▶ Unmodifiable and immutable collections
- ▶ Introduction to concurrent building-blocks
- ▶ Details about concurrent collections