

Collections  
 concurrent collections  
 Producer-Consumer pattern  
 synchronizers  
 Summary building blocks

# Building blocks

Ferd van Odenhoven  
 Pieter van den Hombergh

Fontys Hogeschool voor Techniek en Bedrijfsmanagement

4 december 2008



Collections  
 concurrent collections  
 Producer-Consumer pattern  
 synchronizers  
 Summary building blocks

- ① Collections
- ② concurrent collections
- ③ Producer-Consumer pattern
  - Indexing service example
  - Social behaviour between objects and threads
- ④ synchronizers
  - Latches
  - FutureTask
  - Intermezzo, Taming the Exception
  - Semaphores
  - Barriers
  - The four Memoizers
- ⑤ Summary building blocks



Collections  
 concurrent collections  
 Producer-Consumer pattern  
 synchronizers  
 Summary building blocks

## Intro Buildingblocks

Up to now we discussed:

- threadsafety:
  - atomicity,
  - race conditions,
  - using locks.
- how to share objects:
  - visibility and its relation to locking,
  - how to obtain threadconfinement
  - immutability
  - safe publication
- how to compose your objects for concurrency:
  - making your class threadsafe
  - requirements for synchronization and state of an object
  - using encapsulation
  - java monitor pattern
  - delegating thread safety
  - publishing underlying state variables



- Collections
- concurrent collections
- Producer-Consumer pattern
- synchronizers
- Summary building blocks

### Building Blocks

We will see new **Barriers** instead of old ones:



FvO,PvdH/FHTBM

Building blocks

4 december 2008

4/40

- Collections
- concurrent collections
- Producer-Consumer pattern
- synchronizers
- Summary building blocks

### Building Blocks

We will see **Semaphores** but not for trains:



FvO,PvdH/FHTBM

Building blocks

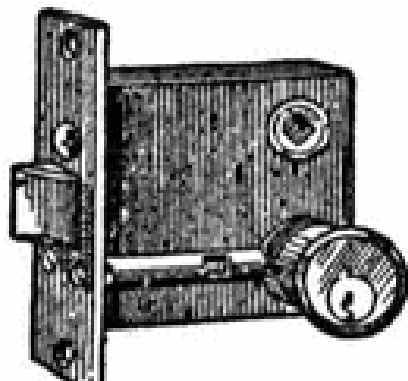
4 december 2008

5/40

- Collections
- concurrent collections
- Producer-Consumer pattern
- synchronizers
- Summary building blocks

### Building Blocks

We will see **Latches** but not these:



FvO,PvdH/FHTBM

Building blocks

4 december 2008

6/40

## Synchronized Collections

Some of the older (jdk2) collections (e.g. Vector) are synchronized and thus threadsafe, but there are problems: with **compound actions**. See for example the *put-if-absent* example of the previous chapter.

These collections keep their integrity: but multiple threads calling these synchronized methods may be confronted with unpredictable states. See figure 5.1 in the book and Listing 5.1.



## Synchronized Collections, cont'd

```

public static Object getLast(Vector list) {
    int lastIndex = list.size() - 1;
    return list.get(lastIndex);
}

public static void deleteLast(Vector list) {
    int lastIndex = list.size() - 1;
    list.remove(lastIndex);
}

```



## A safe Vector helper

- These synchronized collections commit to a synchronization policy that supports so called *client-side locking* (we saw it in the previous chapter: Listings 4.14-15)
- So we can safely lock on the list, See Listing 5.2:

```

public static Object getLast(Vector list) {
    synchronized (list) {
        int lastIndex = list.size() - 1;
        return list.get(lastIndex);
    }
}

public static void deleteLast(Vector list) {
    synchronized (list) {
        int lastIndex = list.size() - 1;
        list.remove(lastIndex);
    }
}

```



- Collections
- concurrent collections
- Producer-Consumer pattern
- synchronizers
- Summary building blocks

## Iterating a collection

The next code-snippet might lead to an `ArrayOutOfBoundsException`, although the collection is threadsafe!

```
for (int i=0; i<vector.size(); i++)
    doSomething(vector.get(i));
```

Which could be prevented by holding the Vector lock during the iteration:

```
synchronized(vector) {
    for (int i=0; i<vector.size(); i++)
        doSomething(vector.get(i));
}
```

But do we really want that???



- Collections
- concurrent collections
- Producer-Consumer pattern
- synchronizers
- Summary building blocks

## ConcurrentModificationException

- The newer collection classes still have a problem:
- By means of an explicit Iterator or via a for-each loop we can bump into a concurrent modification of the collection.
- These collections are what one calls: *fail-fast*:
- if the collection changed since beginning the iteration, an unchecked exception is thrown: an `ConcurrentModificationException`.
- Locking the collection during iteration is often undesirable: loss of performance, possible deadlock, scalability problems.
- Cloning the collection could be a solution.



- Collections
- concurrent collections
- Producer-Consumer pattern
- synchronizers
- Summary building blocks

## Hidden Iterators

- Use locking everywhere a shared collection might be iterated.
- See example on the next slide: `HiddenIterator.java`
- The string concatenation:
 

```
System.out.println("DEBUG: added ten elements to "+ set);
```

 gets returned by the compiler into a call to `StringBuilder.append(Object)`, which in turn invokes the collection's `toString()` method and then the iteration over all objects in the collection starts.
- There is a hidden iterator and a `ConcurrentModificationException` could be thrown.
- `HiddenIterator` is not thread-safe: a lock on set should be used.



Collections  
 concurrent collections  
 Producer-Consumer pattern  
 synchronizers  
 Summary building blocks

## HiddenIterator class

```
public class HiddenIterator {
  @GuardedBy("this")
  private final Set<Integer> set = new HashSet<Integer>();

  public synchronized void add(Integer i) {
    set.add(i);
  }

  public synchronized void remove(Integer i) {
    set.remove(i);
  }

  public void addTenThings() {
    Random r = new Random();
    for (int i = 0; i < 10; i++)
      add(r.nextInt());
    System.out.println("DEBUG: added ten elements to " + set);
  }
}
```



Collections  
 concurrent collections  
 Producer-Consumer pattern  
 synchronizers  
 Summary building blocks

## Hidden Iterators-2

Indirect iteration is also invoked by the methods:

- hashCode and equals
- containsAll, removeAll and retainAll

could throw a ConcurrentModificationException



Collections  
**concurrent collections**  
 Producer-Consumer pattern  
 synchronizers  
 Summary building blocks

## Concurrent Collections

- The synchronized collections suffer from poor concurrency: the price paid for synchronizing all access to it.
- The concurrent collections are designed for multiple access from multiple threads.
  - ConcurrentHashMap replaces hash-based implementations of synchronized Map,
  - CopyOnWriteArrayList replaces synchronized implementations of List
- Support is provided for actions such as put-if-absent, replace and conditional remove.
- New type Queue with implementations: ConcurrentLinkedQueue and PriorityQueue



## Concurrent Collections

- New type `BlockingQueue` extends `Queue` with implementations: `ArrayBlockingQueue`, `DelayQueue`, `LinkedBlockingQueue`, `PriorityBlockingQueue`, `SynchronousQueue`

### Tips

- 1 Replacing synchronized collections with concurrent collections can offer dramatic scalability improvements with little risk.
- 2 `BlockingQueue`'s are extremely useful in producer-consumer designs.



## ConcurrentHashMap

- `ConcurrentHashMap` (among others) have an improved iterator that does not throw a `ConcurrentModificationException` so that a lock on the entire collection is no longer necessary.
- *weakly consistent* instead of *fail fast*: tolerates concurrent modification
- traverses elements as they existed when the iterator was created; e.g. `size` method could give out of date (stale) value.
- this is to improve important methods as `put`, `get`, `containsKey` and `remove`.



## CopyOnWriteArrayList

- `CopyOnWriteArrayList` eliminates the need to lock or copy the collection during iteration.
- publish your immutable objects properly and you have thread safety
- if the collection changes: an new copy is created and published
- not efficient if there are many modifications in a large collection



Collections  
 concurrent collections  
**Producer-Consumer pattern**  
 synchronizers  
 Summary building blocks

Indexing service example  
 Social behaviour between objects and threads

## Producers and consumers

- Producers and consumers are very common in programs.
- Usually some kind of buffering is involved between P and C.
- Very often P and C swap roles in another part of the program like in: P produces full buffer elements and C produces empty buffer places. There is a mutual interdependence.
- The buffer can be implemented as a queue.



FvO,PvdH/FHTBM

Building blocks

4 december 2008

19/40

Collections  
 concurrent collections  
**Producer-Consumer pattern**  
 synchronizers  
 Summary building blocks

Indexing service example  
 Social behaviour between objects and threads

## Blocking Queues to manage workload

### Bounded queues

Bounded queues are a powerful resource management tool to build reliable applications; they make a program more robust to overload by throttling activities that threaten to produce more than can be handled.

- Build resource management into your design early using blocking queues – it is a lot easier to do this at the start then to put it in later.



FvO,PvdH/FHTBM

Building blocks

4 december 2008

20/40

Collections  
 concurrent collections  
**Producer-Consumer pattern**  
 synchronizers  
 Summary building blocks

Indexing service example  
 Social behaviour between objects and threads

## Indexing service

```

30 public class IndexingService {
31     private static final int CAPACITY = 1000;
32     private static final File POISON = new File("");
33     private final IndexerThread consumer = new IndexerThread();
34     private final CrawlerThread producer = new CrawlerThread();
35     private final BlockingQueue<File> queue;
36     private final FileFilter fileFilter;
37     private final File root;
38
39     public IndexingService(File root, final FileFilter fileFilter)
40         this.root = root;
41         this.queue = new LinkedBlockingQueue<File>(CAPACITY);
42         this.fileFilter = new FileFilter() {
43             public boolean accept(File f) {
44                 return f.isDirectory() || fileFilter.accept(f);
45             }
46         };
47     }

```



FvO,PvdH/FHTBM

Building blocks

4 december 2008

21/40

Collections  
concurrent collections  
**Producer-Consumer pattern**  
synchronizers  
Summary building blocks

**Indexing service example**  
Social behaviour between objects and threads

## Indexing service

```

45     private boolean alreadyIndexed(File f) {
46         return false;
47     }
48
49     class CrawlerThread extends Thread {
50         public void run() {
51             try {
52                 crawl(root);
53             } catch (InterruptedException e) { /* fall through */
54             } finally {
55                 while (true) {
56                     try {
57                         queue.put(POISON);
58                         break;
59                     } catch (InterruptedException e1) { /* retry */
60                     }
61                 }
62             }
63         }

```



FvO,PvdH/FHTBM

Building blocks

4 december 2008

22/40

Collections  
concurrent collections  
**Producer-Consumer pattern**  
synchronizers  
Summary building blocks

**Indexing service example**  
Social behaviour between objects and threads

## Indexing service

```

61     private void crawl(File root) throws InterruptedException
62     {
63         File[] entries = root.listFiles(fileFilter);
64         if (entries != null) {
65             for (File entry : entries) {
66                 if (entry.isDirectory())
67                     crawl(entry);
68                 else if (!alreadyIndexed(entry))
69                     queue.put(entry);
70             }
71         }

```



FvO,PvdH/FHTBM

Building blocks

4 december 2008

23/40

Collections  
concurrent collections  
**Producer-Consumer pattern**  
synchronizers  
Summary building blocks

**Indexing service example**  
Social behaviour between objects and threads

## Indexing service

```

69     class IndexerThread extends Thread {
70         public void run() {
71             try {
72                 while (true) {
73                     File file = queue.take();
74                     if (file == POISON)
75                         break;
76                     else
77                         indexFile(file);
78                 }
79             } catch (InterruptedException consumed) {
80             }
81         }
82
83         public void indexFile(File file) {
84             /*...*/
85         };

```



FvO,PvdH/FHTBM

Building blocks

4 december 2008

24/40

## Indexing service

```

83     public void start() {
84         producer.start();
85         consumer.start();
86     }
87
88     public void stop() {
89         producer.interrupt();
90     }
91
92     public void awaitTermination() throws InterruptedException {
93         consumer.join();
94     }
  
```



## Safe handing over

- Objects often present a unit of work.
- The produce consumer use the queue (even of length 0, the kind used in a *rendevouz*) to safely hand over objects.
- During the processing, the objects that are taken from or put into the queue are thread local to either producer or consumer.
- Those objects themselves are then protected by the threadlocality and need not be designed threadsafe and locking can be avoided in those objects.



## Helpers, or: Stealing work

- Java 6 introduces the double ended queue **Deque** and the **BlockingQueue**.
- The Deque allows 'work' to be put and taken from both ends of a queue.
- This allows workload sharing between consumers that all have their own work queue, but may help another consumer with that one's work if the own queue is empty. This is called work stealing.
- Quiz: Why is taking work from *The other end* a good idea, concurrency wise?



Collections  
 concurrent collections  
 Producer-Consumer pattern  
**synchronizers**  
 Summary building blocks

**Latches**  
**FutureTask**  
 Intermezzo, Taming the Exception  
 Semaphores  
 Barriers  
 The four Memoizers

## Synchronizers

A *synchronizer* is any object that coordinates the control flow of threads based on its state.



FvO,PvdH/FHTBM

Building blocks

4 december 2008

28/40

Collections  
 concurrent collections  
 Producer-Consumer pattern  
**synchronizers**  
 Summary building blocks

**Latches**  
**FutureTask**  
 Intermezzo, Taming the Exception  
 Semaphores  
 Barriers  
 The four Memoizers

## Latches can be used to

- Ensuring that a computation does not proceed until the resources it needs to have are initialized.
- Ensuring that a service does not start until other services on which it is dependent have started.
- Wait until all parties involved in an activity, for instance players in a multiplayer game, are ready to proceed.
- And of course, they can be used in concurrency exercises.



, for instance to make sure that threads indeed start together in tests.

Note that latches can be fired only once.



FvO,PvdH/FHTBM

Building blocks

4 december 2008

29/40

Collections  
 concurrent collections  
 Producer-Consumer pattern  
**synchronizers**  
 Summary building blocks

**Latches**  
**FutureTask**  
 Intermezzo, Taming the Exception  
 Semaphores  
 Barriers  
 The four Memoizers

## Postponing work into the future...

- Future tasks are made up of **Future** and **Callable**, the resultbearing relative of **Runnable**.
- There are several ways to complete:
  - Normal completion
  - cancelation
  - and exception.
- Once a **FutureTask** is completed in cannot be restarted.
- **Future.get()** returns the result immediately if 'the future is here' (Task is completed) and
- Blocks if the task is not complete yet.



FvO,PvdH/FHTBM

Building blocks

4 december 2008

30/40

Collections  
 concurrent collections  
 Producer-Consumer pattern  
**synchronizers**  
 Summary building blocks

Latches  
**FutureTask**  
 Intermezzo, Taming the Exception  
 Semaphores  
 Barriers  
 The four Memoizers

## Example Preloader

```

private final FutureTask<ProductInfo> future =
    new FutureTask<ProductInfo>(new Callable<ProductInfo>() {
        public ProductInfo call() throws DataLoadException {
            return loadProductInfo();
        }
    });
private final Thread thread = new Thread(future);

public void start() { thread.start(); }

public ProductInfo get()
    throws DataLoadException, InterruptedException {
    try {
        return future.get();
    } catch (ExecutionException e) {
        Throwable cause = e.getCause();
        if (cause instanceof DataLoadException)
            throw (DataLoadException) cause;
        else
            throw LaunderThrowable.launderThrowable(cause);
    }
}

```



FvO,PvdH/FHTBM

Building blocks

4 december 2008

31/40

Collections  
 concurrent collections  
 Producer-Consumer pattern  
**synchronizers**  
 Summary building blocks

Latches  
 FutureTask  
**Intermezzo, Taming the Exception**  
 Semaphores  
 Barriers  
 The four Memoizers

## Coercing an unchecked Throwable

into a RuntimeException

```

public class LaunderThrowable {

    /**
     * Coerce an unchecked Throwable to a RuntimeException
     * <p/>
     * If the Throwable is an Error, throw it; if it is a
     * RuntimeException return it, otherwise throw IllegalStateException
     */
    public static RuntimeException launderThrowable(Throwable t) {
        if (t instanceof RuntimeException)
            return (RuntimeException) t;
        else if (t instanceof Error)
            throw (Error) t;
        else
            throw new IllegalStateException("Not unchecked", t);
    }
}

```



FvO,PvdH/FHTBM

Building blocks

4 december 2008

32/40

Collections  
 concurrent collections  
 Producer-Consumer pattern  
**synchronizers**  
 Summary building blocks

Latches  
 FutureTask  
 Intermezzo, Taming the Exception  
**Semaphores**  
 Barriers  
 The four Memoizers

## Semaphores: A very early synchronization concept

- A (counting) semaphore manages a set of permits
- As long as permits are available (count > 0) activities can acquire one and continue into the region that requires the permit.
- Binary semaphore have just one permit and are also called and used as mutual exclusion devices or **Mutexes**.
- Example is managing a pool of say database connections. The number of initial permits to the pool is equal to the number of those resources
- The activity that needs a resource tries to get a permit and *blocks* if none is available.
- The counting semaphore can help to make any collection into a bounded set. (Why not have a hashmap with a bounded capacity: here is how to make one.)



FvO,PvdH/FHTBM

Building blocks

4 december 2008

33/40

Collections	Latches
concurrent collections	FutureTask
Producer-Consumer pattern	Intermezzo, Taming the Exception
<b>synchronizers</b>	Semaphores
Summary building blocks	<b>Barriers</b>
	The four Memoizers

## Barriers

- The difference with Latches is that Latches wait for events and Barriers wait for other Threads
- Meet us at StarBucks at 18:00
- If you arrive, call `await()`, to wait for the others. Once all (set by a creation count) have arrived, all proceed
- Of course, if one spills coffee while waiting, all will receive a `BrokenBarrierException`
- The `await` also is the jury on who arrived earliest, so that Thread may get special privileges (like to say what to do next)
- Barriers are usefull in simulations



Collections	Latches
concurrent collections	FutureTask
Producer-Consumer pattern	Intermezzo, Taming the Exception
<b>synchronizers</b>	Semaphores
Summary building blocks	<b>Barriers</b>
	<b>The four Memoizers</b>

## The core of Memoizer1

This is poor concurrency..



```
public synchronized V compute(A arg) throws InterruptedException {
    V result = cache.get(arg);
    if (result == null) {
        result = c.compute(arg);
        cache.put(arg, result);
    }
    return result;
}
```



Collections	Latches
concurrent collections	FutureTask
Producer-Consumer pattern	Intermezzo, Taming the Exception
<b>synchronizers</b>	Semaphores
Summary building blocks	<b>Barriers</b>
	<b>The four Memoizers</b>

## The core of Memoizer2

This one does not prevent double work .



```
public class Memoizer2 <A, V> implements Computable<A, V> {
    private final Map<A, V> cache = new ConcurrentHashMap<A, V>();
    private final Computable<A, V> c;

    public Memoizer2(Computable<A, V> c) {
        this.c = c;
    }

    public V compute(A arg) throws InterruptedException {
        V result = cache.get(arg);
        if (result == null) {
            result = c.compute(arg);
            cache.put(arg, result);
        }
        return result;
    }
}
```



Collections  
 concurrent collections  
 Producer-Consumer pattern  
**synchronizers**  
 Summary building blocks

Latches  
 FutureTask  
 Intermezzo, Taming the Exception  
 Semaphores  
 Barriers  
**The four Memoizers**

## The core of Memoizer3

Still a tiny window for double calculation, using a



```

ConcurrentHashMap<A, Future<V>>();

public V compute(final A arg) throws InterruptedException {
    Future<V> f = cache.get(arg);
    if (f == null) {
        Callable<V> eval = new Callable<V>() {
            public V call() throws InterruptedException {
                return c.compute(arg);
            }
        };
        FutureTask<V> ft = new FutureTask<V>(eval);
        f = ft;
        cache.put(arg, ft);
        ft.run(); // call to c.compute happens here
    }
    try {
        return f.get();
    } catch (ExecutionException e) {
        throw LaunderThrowable.laundryThrowable(e.getCause());
    }
}

```



Collections  
 concurrent collections  
 Producer-Consumer pattern  
**synchronizers**  
 Summary building blocks

Latches  
 FutureTask  
 Intermezzo, Taming the Exception  
 Semaphores  
 Barriers  
**The four Memoizers**

## Use putIfAbsent of ConcurrentHashMap

```

public V compute(final A arg) throws InterruptedException {
    while (true) {
        Future<V> f = cache.get(arg);
        if (f == null) {
            Callable<V> eval = new Callable<V>() {
                public V call() throws InterruptedException {
                    return c.compute(arg);
                }
            };
            FutureTask<V> ft = new FutureTask<V>(eval);
            f = cache.putIfAbsent(arg, ft);
            if (f == null) {
                f = ft;
                ft.run();
            }
        }
        try {
            return f.get();
        } catch (CancellationException e) {
            cache.remove(arg, f);
        } catch (ExecutionException e) {
            throw LaunderThrowable.laundryThrowable(e.getCause());
        }
    }
}

```



Collections  
 concurrent collections  
 Producer-Consumer pattern  
 synchronizers  
**Summary building blocks**

## Summary I

- It is the mutable state, stupid
- Make fields final unless they *need* to be mutable
- Immutable object are automatically threadsafe
- Encapsulation make managing complexity practical
- Guard each mutable with a lock
- Guard al participants of an invariant with the *same* lock
- Hold locks for the duration of compound actions



