Toc

Building Blocks HOM Collections Synchronized Collections Concurrent modification Hidden Iterators Concurrent Collections Producer-Consumer pattern Indexing service example Social behaviour between objects and threads Synchronizers Synchronization building blocks FutureTask Intermezzo, Taming the Exception Semaphores Barriers Per formant and safe caching The four Memoizers 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

Building Blocks HOM Collections Synchronized Collections Concurrent modification Hidden Iterators Concurrent Collections Producer-Consumer pattern Indexing service example Social behaviour between objects and threads Synchronizers Synchronization building blocks FutureTask Intermezzo, Taming the Exception Semaphores Barriers Per formant and safe caching The four Memoizers Summary building blocks

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.

Building Blocks HOM Collections Synchronized Collections Concurrent modification Hidden Iterators Concurrent Collections Producer-Consumer pattern Indexing service example Social behaviour between objects and threads Synchronizers Synchronization building blocks FutureTask Intermezzo, Taming the Exception Semaphores Barriers Per formant and safe caching The four Memoizers Summary building blocks

Synchronized Collections, cont'd

```java
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);
}
```

Building Blocks HOM Collections Synchronized Collections Concurrent modification Hidden Iterators Concurrent Collections Producer-Consumer pattern Indexing service example Social behaviour between objects and threads Synchronizers Synchronization building blocks FutureTask Intermezzo, Taming the Exception Semaphores Barriers Per formant and safe caching The four Memoizers Summary building blocks

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);
    }
}
```


Iterating a collection

The next code-snippet might lead to an `ArrayOutOfBoundException`, 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???

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 is called *fail-fast*:
  - if the collection changes since beginning an iteration, an unchecked exception is thrown: an `ConcurrentModificationException`.
    (**from the vector java doc**) Fail-fast iterators throw ConcurrentModificationException on a best-effort basis: the fail-fast behavior of iterators should only be used to detect bugs.
- Locking the collection during iteration is often undesirable: loss of performance, possible deadlock, scalability problems.
- Cloning the collection could be a solution.

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.

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 " +
    }
}
```


Hidden Iterators-2

Indirect iteration is also invoked by the methods:

- hashCode and equals
- containsAll, removeAll and retainAll

they could all throw a `ConcurrentModificationException`

Concurrent Collections

- The synchronized colections 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`
  - `CopyOnWriteArrayList` replaces synchronized implementations of `List`
  - `ConcurrentSkipListMap<K,V>` replaces LinkedHashMap
- Support is provided for actions such as put-if-absent, replace and conditional remove.
- New type `Queue` with implementations: `ConcurrentLinkedQueue` and `PriorityQueue`
- The concurrent maps have set siblings, sharing implementation quite intimately.

Concurrent Collections

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

Tips

1. Replacing synchronized collections with concurent 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 iterration.
- 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

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.

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.

Indexing service


```java
public class IndexingService {
    private static final int CAPACITY = 1000;
    private static final File POISON = new File("");
    private final IndexerThread consumer = new IndexerThread();
    private final CrawlerThread producer = new CrawlerThread();
    private final BlockingQueue<File> queue;
    private final FileFilter fileFilter;
    private final File root;

    public IndexingService(File root, final FileFilter fileFilter) {
        this.root = root;
        this.queue = new LinkedBlockingQueue<File>(CAPACITY);
        this.fileFilter = new FileFilter() {
            public boolean accept(File f) {
                return f.isDirectory() || fileFilter.accept(f);
            }
        };
    }
```


Indexing service


```java
    private boolean alreadyIndexed(File f) {
        return false;
    }

    class CrawlerThread extends Thread {
        public void run() {
            try {
                crawl(root);
            } catch (InterruptedException e) { /* fall through */
            } finally {
                while (true) {
                    try {
                        queue.put(POISON);
                        break;
                    } catch (InterruptedException e1) { /* retry */
                    }
                }
            }
        }
```


Indexing service


```java
        private void crawl(File root) throws InterruptedException {
            File[] entries = root.listFiles(fileFilter);
            if (entries != null) {
                for (File entry : entries) {
                    if (entry.isDirectory())
                        crawl(entry);
                    else if (!alreadyIndexed(entry))
                        queue.put(entry);
                }
            }
        }
```


Indexing service


```java
    class IndexerThread extends Thread {
        public void run() {
            try {
                while (true) {
                    File file = queue.take();
                    if (file == POISON)
                        break;
                    else
                        indexFile(file);
                }
            } catch (InterruptedException consumed) {
            }
        }

        public void indexFile(File file) {
            /*...*/
        };
```

Indexing service


```
    public void start() {
        producer.start();
        consumer.start();
    }

    public void stop() {
        producer.interrupt();
    }

    public void awaitTermination() throws InterruptedException {
        consumer.join();
    }
```


Safe handing over

Building Blocks
HOM
Collections
Synchronized Collections
Concurrent modification
Hidden Iterators
Concurrent Collections
Producer-Consumer pattern
Indexing service example
Social behaviour between objects and threads
Synchronizers
Synchronization building blocks
FutureTask
Intermezzo, Taming the Exception
Semaphores
Barriers
Per formant and safe caching
The four Memoizers
Summary building blocks

- Objects often present a unit of work.
- The producer-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

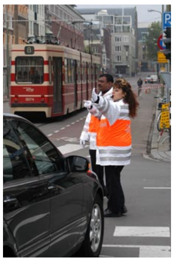
Building Blocks
HOM
Collections
Synchronized Collections
Concurrent modification
Hidden Iterators
Concurrent Collections
Producer-Consumer pattern
Indexing service example
Social behaviour between objects and threads
Synchronizers
Synchronization building blocks
FutureTask
Intermezzo, Taming the Exception
Semaphores
Barriers
Per formant and safe caching
The four Memoizers
Summary building blocks

- Java 6 introduces the interfaces double ended queue `Deque` and `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?

Synchronizers

Building Blocks
HOM
Collections
Synchronized Collections
Concurrent modification
Hidden Iterators
Concurrent Collections
Producer-Consumer pattern
Indexing service example
Social behaviour between objects and threads
Synchronizers
Synchronization building blocks
FutureTask
Intermezzo, Taming the Exception
Semaphores
Barriers
Per formant and safe caching
The four Memoizers
Summary building blocks

A *synchronizer* is any object that coordinates the control flow of threads based on its state.
Examples are **Semaphores**, **Latches** and **Barriers** and **Mutex**es.



Building Blocks: Barriers

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

A (Cyclic)Barrier
A synchronization aid that allows a set of threads to all wait for each other to reach a common barrier point. CyclicBarriers are useful in programs involving a fixed sized party of threads that must occasionally wait for each other. The barrier is called cyclic because it can be re-used after the waiting threads are released. CyclicBarrier javadoc

Figure: No horse racing here.

Building Blocks: Semaphores

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

A counting semaphore
Conceptually, a semaphore maintains a set of permits. Each acquire() blocks if necessary until a permit is available, and then takes it. Each release() adds a permit, potentially releasing a blocking acquirer. However, no actual permit objects are used; the Semaphore just keeps a count of the number available and acts accordingly.
Counting Semaphore javadoc

Figure: No horse racing here.

Building Blocks
HOM
Collections
Synchronized Collections
Concurrent modification
Hidden Iterators
Concurrent Collections
Producer-Consumer pattern
Indexing service example
Social behaviour between objects and threads
Synchronizers
Synchronization building blocks
FutureTask
Intermezzo, Taming the Exception
Semaphores
Barriers
Per formant and safe caching
The four Memoizers
Summary building blocks

Building Blocks: Latches

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

CountDownLatch
A synchronization aid that allows one or more threads to wait until a set of operations being performed in other threads completes.
A CountDownLatch is initialised with a given count. The await methods block until the current count reaches zero due to invocations of the countDown() method, after which all waiting threads are released and any subsequent invocations of await return immediately. This is a one-shot phenomenon – the count cannot be reset. If you need a version that resets the count, consider using a CyclicBarrier. CountDownLatch javadoc

Figure: A more permanent latch.

Building Blocks
HOM
Collections
Synchronized Collections
Concurrent modification
Hidden Iterators
Concurrent Collections
Producer-Consumer pattern
Indexing service example
Social behaviour between objects and threads
Synchronizers
Synchronization building blocks
FutureTask
Intermezzo, Taming the Exception
Semaphores
Barriers
Per formant and safe caching
The four Memoizers
Summary building blocks

Science fiction?

Java 7 introduces the `Phaser`

A reusable synchronization barrier, similar in functionality to CyclicBarrier and CountDownLatch but supporting more flexible usage.
Phaser
Phaser javadoc

Figure: Any Klingons around?.

Building Blocks
HOM
Collections
Synchronized Collections
Concurrent modification
Hidden Iterators
Concurrent Collections
Producer-Consumer pattern
Indexing service example
Social behaviour between objects and threads
Synchronizers
Synchronization building blocks
FutureTask
Intermezzo, Taming the Exception
Semaphores
Barriers
Per formant and safe caching
The four Memoizers
Summary building blocks

Latches can be used to...

- Ensure that a computation does not proceed until the resources it needs are initialised,
- Ensure that a service does not start until other services on which it depends 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.

Building Blocks
HOM
Collections
Synchronized Collections
Concurrent modification
Hidden Iterators
Concurrent Collections
Producer-Consumer pattern
Indexing service example
Social behaviour between objects and threads
Synchronizers
Synchronization building blocks
FutureTask
Intermezzo, Taming the Exception
Semaphores
Barriers
Per formant and safe caching
The four Memoizers
Summary building blocks

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
  - cancellation
  - 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.

Building Blocks
HOM
Collections
Synchronized Collections
Concurrent modification
Hidden Iterators
Concurrent Collections
Producer-Consumer pattern
Indexing service example
Social behaviour between objects and threads
Synchronizers
Synchronization building blocks
FutureTask
Intermezzo, Taming the Exception
Semaphores
Barriers
Per formant and safe caching
The four Memoizers
Summary building blocks

Example Preloader

```java
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);
    }
}
```

Building Blocks
HOM
Collections
Synchronized Collections
Concurrent modification
Hidden Iterators
Concurrent Collections
Producer-Consumer pattern
Indexing service example
Social behaviour between objects and threads
Synchronizers
Synchronization building blocks
FutureTask
Intermezzo, Taming the Exception
Semaphores
Barriers
Per formant and safe caching
The four Memoizers
Summary building blocks

Coercing an unchecked Throwable

into a RuntimeException

```java
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 IllegalStateExcept
     */
    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);
    }
}
```

Building Blocks
HOM
Collections
Synchronized Collections
Concurrent modification
Hidden Iterators
Concurrent Collections
Producer-Consumer pattern
Indexing service example
Social behaviour between objects and threads
Synchronizers
Synchronization building blocks
FutureTask
Intermezzo, Taming the Exception
Semaphores
Barriers
Per formant and safe caching
The four Memoizers
Summary building blocks

Semaphores: A very early synchronization concept

- A (counting) semaphore manages a set of permits
- As long as permits are available (count > 0) activities can aquire 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 **Mutex**es.
- 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.)

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

Building Blocks
HOM
Collections
Synchronized Collections
Concurrent modification
Hidden Iterators
Concurrent Collections
Producer-Consumer pattern
Indexing service example
Social behaviour between objects and threads
Synchronizers
Synchronization building blocks
FutureTask
Intermezzo, Taming the Exception
Semaphores
Barriers
Per formant and safe caching
The four Memoizers
Summary building blocks

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;
}
```

Building Blocks
HOM
Collections
Synchronized Collections
Concurrent modification
Hidden Iterators
Concurrent Collections
Producer-Consumer pattern
Indexing service example
Social behaviour between objects and threads
Synchronizers
Synchronization building blocks
FutureTask
Intermezzo, Taming the Exception
Semaphores
Barriers
Per formant and safe caching
The four Memoizers
Summary building blocks

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;
    }
}
```

Building Blocks
HOM
Collections
Synchronized Collections
Concurrent modification
Hidden Iterators
Concurrent Collections
Producer-Consumer pattern
Indexing service example
Social behaviour between objects and threads
Synchronizers
Synchronization building blocks
FutureTask
Intermezzo, Taming the Exception
Semaphores
Barriers
Per formant and safe caching
The four Memoizers
Summary building blocks

The core of Memoizer3

Still a tiny window for double calculation, using a ConcurrentHashMap<A, Future<V>>();

```java
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.launderThrowable(e.getCause());
    }
}
```


Use putIfAbsent of ConcurrentHashMap

```java
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.launderThrowable(e.getCause());
        }
    }
}
```

Building Blocks
HOM
Collections
Synchronized Collections
Concurrent modification
Hidden Iterators
Concurrent Collections
Producer-Consumer pattern
Indexing service example
Social behaviour between objects and threads
Synchronizers
Synchronization building blocks
FutureTask
Intermezzo, Taming the Exception
Semaphores
Barriers
Per formant and safe caching
The four Memoizers
Summary building blocks

Summary I

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

Building Blocks
HOM
Collections
Synchronized Collections
Concurrent modification
Hidden Iterators
Concurrent Collections
Producer-Consumer pattern
Indexing service example
Social behaviour between objects and threads
Synchronizers
Synchronization building blocks
FutureTask
Intermezzo, Taming the Exception
Semaphores
Barriers
Per formant and safe caching
The four Memoizers
Summary building blocks

Summary II

- Any program that accesses a mutable from multiple threads *without* synchronization is a *broken* program.
- Never rely on clever reasoning on why you do not need synchronization in those situations
- Include thread safety into the design process – or explicitely document that your class is (intentionally) not thread-safe
- Document your synchronization policy

Building Blocks
HOM
Collections
Synchronized Collections
Concurrent modification
Hidden Iterators
Concurrent Collections
Producer-Consumer pattern
Indexing service example
Social behaviour between objects and threads
Synchronizers
Synchronization building blocks
FutureTask
Intermezzo, Taming the Exception
Semaphores
Barriers
Per formant and safe caching
The four Memoizers
Summary building blocks