Synchronized Collections in java.util.

- Vector
- Hashtable
- `Collections.synchronizedCollection(collection)`
- `Collections.synchronizedSet(set)`
- `Collections.synchronizedSortedSet(sortedSet)`
- `Collections.synchronizedList(list)`
- `Collections.synchronizedMap(map)`

They encapsulate their state and synchronize every method. They commit to explicit support for client-side locking through their own intrinsic lock.
Compound Actions are not Atomic

@NotThreadSafe

public class UnsafeVectorHelpers {

    public static <T> T getLast(Vector<T> list) {
        int lastIndex = list.size() - 1; // read-modify
        return list.get(lastIndex); // write
    }

    public static <T> void deleteLast(Vector<T> list) {
        int lastIndex = list.size() - 1; // read-modify
        list.remove(lastIndex); // write
    }
}

In a concurrent setting, this might throw an
ArrayIndexOutOfBoundsException
@ThreadSafe
public class SafeVectorHelpers {
    public static <T> T getLast(Vector<T> list) {
        synchronized (list) {
            int lastIndex = list.size() - 1;
            return list.get(lastIndex);
        }
    }

    public static <T> void deleteLast(Vector<T> list) {
        synchronized (list) {
            int lastIndex = list.size() - 1;
            list.remove(lastIndex);
        }
    }
}
Hidden Compound Actions in Iteration

In a concurrent setting, the subsequent code might throw an ArrayIndexOutOfBoundsException:

```java
@NotThreadSafe
for (int i = 0; i < vector.size(); i++) // check
doSomething(vector.get(i)); // then-act
```

How to fix it:

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

However, this solution prevents other threads from accessing the vector for the whole duration of the iteration, which is undesirable.
This exception is thrown by iterators on collections, if the original collection gets modified before the iteration ends. It occurs in a sequential setting as well:

```java
for (E element: list)
    if (isBad(element))
        list.remove(element); // ConcurrentModificationException
```

In a concurrent setting, the exception might be thrown in much more surprising situations:

```java
@NotThreadSafe
for (E element: vector) // may throw ConcurrentModificationException
    System.out.println(element);
```
The problem with iteration is that it is a hidden check-then-act operation:

```java
@NotThreadSafe
for (E element: vector) // check
    System.out.println(element); // then-act
```

A way to fix it:

```java
@ThreadSafe
synchronized (vector) {
    for (E element: vector)
        System.out.println(element);
}
```

However, this solution prevents other threads from accessing the vector for the whole duration of the iteration, which is undesirable.
Beware of Implicit Iteration

@NotThreadSafe
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());
    // missing synchronization below:
    // this may throw a ConcurrentModificationException
    System.out.println("DEBUG: added ten elements to "+ set);
}
}
Other Implicit Uses of Iteration on Collections

- `hashCode()` and `equals()`, possibly called if a collection is used as an element of another collection
- `containsAll()`, `removeAll()`, `retainAll()`
- Constructors that take a collection as parameter and build a copy

Always synchronize on the shared mutable collection before such operations
Concurrent Collections

Synchronized collections: Vector, Hashtable, synchronizedXXX
Thread-safety achieved by serializing all access: poor concurrency

Concurrent collections: ConcurrentHashMap, CopyOnWriteArrayList
Multiple threads can concurrently access the collection in a thread-safe way: high concurrency, slightly increased memory footprint
ConcurrentHashMap implements ConcurrentMap

Synchronized collections synchronize for the full duration of a simple map.get(key) operation, which might take longer than expected. ConcurrentHashMap instead uses lock striping:

- readers can access the map concurrently
- readers and writers can access the map concurrently
- a limited number of writers can modify the map concurrently
- iterators are not fail-fast: they yield a snapshot of the map at the time of their creation and never throw a ConcurrentModificationException
- there is no need to synchronize during iteration
- there is no support for client-side locking
- there are extra atomic check-then-act operations
  - V putIfAbsent(K key, V value)
  - boolean remove(K key, V value)
  - boolean replace(K key, V oldValue, V newValue)
  - V replace(K key, V newValue)
CopyOnWriteArrayList and CopyOnWriteArraySet

They hold a backing collection that is recreated every time the collection gets modified:

- iterators refer to the backing collection at the time of creation, hence they reflect a snapshot of the collection at their creation time, are not fail-safe and never throw a ConcurrentModificationException
- there is no need to synchronize during iteration
- mutative methods are slow

They are the perfect concurrent data structure when modification is rare, while iteration is the predominant operation

- such as for implementing a list of listeners, that must be iterated at each notification
@ThreadSafe
public class VisualComponent {
    private final List<KeyListener> keyListeners = new CopyOnWriteArrayList<>();

    public void addKeyListener(KeyListener listener) {
        keyListeners.add(listener);
    }

    public void removeKeyListener(KeyListener listener) {
        keyListeners.remove(listener);
    }

    public void fireKeyListeners(KeyEvent event) {
        // no need to synchronize here
        for (KeyListener listener: keyListeners)
            listener.keyPressed(event);
    }
}
Blocking Queues

Blocking queues implement a FIFO or priority buffer with methods:

- `put(element)` that adds an element to the queue
  - **bounded** queues: blocks if full
  - **unbounded** queues: never blocks

- `take()` that extracts the first element from the queue
  - if the queue is empty, it blocks

- `offer(element)` that adds an element to the queue
  - **bounded** queues: returns `false` if full
  - **unbounded** queues: equivalent to `put`

- `poll()` that extracts the first element from the queue
  - if the queue is empty, yields `null`

- timed versions of `offer` and `poll`

Beware of unbounded queues that might fill up the memory
Producer/Consumer Pattern

Diagram showing a BlockingQueue with arrows indicating 'Put' from Thread 1 and 'Take' from Thread 2.
private static final int BOUND = 10;
private static final int N_CONSUMERS
    = Runtime.getRuntime().availableProcessors();

public static void startIndexing(File[] roots) {
    BlockingQueue<File> queue = new LinkedBlockingQueue<>(BOUND);

    // start producers
    for (File root: roots)
        new Thread(new FileCrawler(queue, root)).start();

    // start consumers
    for (int i = 0; i < N_CONSUMERS; i++)
        new Thread(new Indexer(queue)).start();
}
The Producer

@ThreadSafe
class FileCrawler implements Runnable {
    private final BlockingQueue<File> fileQueue;
    private final File root;

    public FileCrawler(BlockingQueue<File> fileQueue, File root) {
        this.fileQueue = fileQueue;
        this.root = root;
    }

    public void run() {
        crawl(root);
    }

    private void crawl(File root) {
        File[] entries = root.listFiles();
        if (entries != null)
            for (File entry : entries)
                if (entry.isDirectory())
                    crawl(entry);
                else if (!alreadyIndexed(entry))
                    fileQueue.put(entry);
    }
}
@ThreadSafe
class Indexer implements Runnable {
    private final BlockingQueue<File> queue;

    public Indexer(BlockingQueue<File> queue) {
        this.queue = queue;
    }

    public void run() {
        while (true)
            indexFile(queue.take());
    }

    public void indexFile(File file) {
        // index the file...
    }
}
Cooperative interruption

A thread cannot be stopped. It is only possible to send an interruption request to a thread by calling its `interrupt()` method. This will result in

- a checked `InterruptedException`, if the thread is blocked at a blocking method (such as `queue.take()`)
- its interruption flag being set, otherwise. This can be checked through the `isInterrupted()` method of the thread
Dealing with `InterruptedException`

The simplified code of two slides ago:

```java
class Indexer implements Runnable {
    ...
    public void run() {
        while (true)
            indexFile(queue.take());
    }
}
```

The real code:

```java
class Indexer implements Runnable {
    ...
    public void run() {
        try {
            while (true)
                indexFile(queue.take());
        } catch (InterruptedException e) {
            // propagate back the interruption to the thread running this Runnable
            Thread.currentThread().interrupt();
        }
    }
}
```
Never Swallow an Interruption

Interruptions should not be eaten

class Indexer implements Runnable {
    
    public void run() {
        try {
            while (true)
                indexFile(queue.take());
        }
        catch (InterruptedException e) {}
            // don’t do this at home
    }
}

This means that the request has been ignored. Instead
  - either propagate back the exception
  - or propagate back the interruption request

Interruptions could be swallowed only if you are extending Thread!
A synchronizer is any object that coordinates the control flow of threads based on its state:

- blocking queues
- latches
- semaphores
- barriers
- futures
A **latch** is a synchronizer that can delay the progress of threads until it reaches its **terminal** state.

Class **CountDownLatch** allows threads to wait for a set of events to occur. It is initialized to a positive number, representing the number of events to wait for:

- method **countDown()** decrements the counter
- method **await()** blocks until the counter reaches zero
public class TestHarness {
    public long timeTasks(int nThreads, final Runnable task) throws InterruptedException {
        final CountDownLatch startGate = new CountDownLatch(1);
        final CountDownLatch endGate = new CountDownLatch(nThreads);

        for (int i = 0; i < nThreads; i++)
            new Thread() {
                public void run() {
                    try {
                        startGate.await(); // wait for the GO!
                        try {
                            task.run();
                        } finally {
                            endGate.countDown(); // finished!
                        }
                    } catch (InterruptedException ignored) {}
                }
            }.start();

        long start = System.nanoTime();
        startGate.countDown(); // let all thread start now
        endGate.await(); // wait for the last thread to finish
        return System.nanoTime() - start;
    }
}

Fausto Spoto
Università di Verona, Italy - 25 / 46
A **semaphore** is a synchronizer that controls the number of agents that can access a certain resource.

Class Semaphore is initialized to a positive number, representing the number of available access permits:

- **method acquire()** grabs a permit and blocks if no permit is available.
- **method release()** pushes back a permit.
@ThreadSafe
public class BoundedHashSet<T> {
    private final Set<T> set;
    private final Semaphore sem;

    public BoundedHashSet(int bound) {
        this.set = Collections.synchronizedSet(new HashSet<>());
        this.sem = new Semaphore(bound);
    }
}
public boolean add(T o) throws InterruptedException {
    sem.acquire();
    boolean wasAdded = false;
    try {
        wasAdded = set.add(o);
        return wasAdded;
    } finally {
        if (!wasAdded)
            sem.release();
    }
}

public boolean remove(Object o) {
    boolean wasRemoved = set.remove(o);
    if (wasRemoved)
        sem.release();
    return wasRemoved;
}
A **barrier** is a synchronizer that blocks a group of threads until they have all reached a barrier point, then they proceed.

Class **CyclicalBarrier** is initialized to a positive number $n$, representing the number of threads that should be awaited at the barrier.

- method `await()` blocks until the remaining $n - 1$ have called `await()` as well, at which moment all $n$ threads are allowed to proceed.
public class CellularAutomata {
    private final Board mainBoard;
    private final CyclicBarrier barrier;
    private final Worker[] workers;

    public CellularAutomata(Board board) {
        int count = Runtime.getRuntime().availableProcessors();
        this.mainBoard = board;
        this.barrier = new CyclicBarrier(count, () -> mainBoard.commitNewValues());
        this.workers = new Worker[count];
        for (int i = 0; i < count; i++) // split work between workers
            workers[i] = new Worker(mainBoard.getSubBoard(count, i));
    }

    public void start() {
        // start all workers
        for (Worker worker : workers)
            new Thread(worker).start();
        mainBoard.waitForConvergence();
    }
}
// inner class
private class Worker implements Runnable {
    private final Board board;

    public Worker(Board board) { this.board = board; }

    public void run() {
        while (!board.hasConverged()) {
            // compute the next generation of the values
            for (int x = 0; x < board.getMaxX(); x++)
                for (int y = 0; y < board.getMaxY(); y++)
                    board.setNewValue(x, y, computeValue(x, y));

            try {
                // wait until all other workers have finished
                barrier.await();
            }
            catch (InterruptedException | BrokenBarrierException ex) {
                return; // nobody can interrupt this
            }
        }
    }
}
## Latches, Semaphores and Barriers at a Glance

<table>
<thead>
<tr>
<th><strong>Latch</strong></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>new CountDownLatch(counter &gt;= 0)</code></td>
<td></td>
</tr>
<tr>
<td><code>countDown()</code> decrements the counter</td>
<td></td>
</tr>
<tr>
<td><code>await()</code> blocks until the counter reaches 0</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Semaphore</strong></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>new Semaphore(counter &gt;= 0)</code></td>
<td></td>
</tr>
<tr>
<td><code>acquire()</code> decrements the counter and blocks if it is 0</td>
<td></td>
</tr>
<tr>
<td><code>release()</code> increments the counter</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Barrier</strong></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>new CyclicBarrier(counter &gt;= 0)</code></td>
<td></td>
</tr>
<tr>
<td><code>await()</code> decrements the counter and blocks if it is not 0. If it reaches 0, reset the counter to its initial value and wakes up all blocked threads</td>
<td></td>
</tr>
</tbody>
</table>
Runnable

The specification of a task that does not return any value nor throws any exception

Callable<V>

The specification of a task that can return a value of type V or throw an exception

Future<V>

A pointer to the future result of a computation, not necessarily terminated yet, that eventually will return a value of type V or throw an exception
A Pointer to a Future Result

// specify the task to compute a V as a Callable<V> c
FutureTask<V> future = new FutureTask<>(c);

// build and start your worker
Thread thread = new Thread(future);
thread.start();

// do other things here
// and others things as well
// do more things

// eventually, ask for the result, when you need it
V result = future.get();
Getting the Result of a FutureTask<V>

The statement `V result = future.get()` might have many outcomes:

- the worker has already terminated and computed value \( r \Rightarrow \) assign \( r \) immediately to variable `result` and continue
- the worker has already terminated by throwing an exception \( e \Rightarrow \) throw immediately a new `ExecutionException` with cause \( e \)
- the worker has been cancelled \( \Rightarrow \) throws a new `CancellationException`
- the worker is still working \( \Rightarrow \) block until:
  - either the worker terminates by computing a value \( r \Rightarrow \) wake up, assign \( r \) to `result` and continue
  - or the worker terminates by throwing an exception \( e \Rightarrow \) wake up and throw a new `ExecutionException` with cause \( e \)
  - or the blocked thread gets interrupted \( \Rightarrow \) throw a new `InterruptedException`
public class Preloader {
    private final FutureTask<ProductInfo> future =
        new FutureTask<>(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(cause); // cast cause into unchecked
        }
    }
}
unchecked exceptions: instances of RuntimeException and Error. No need to declare them with throws

checked exceptions: all other exceptions. Must be declared with throws

If we know that t is an unchecked exception, the following code will throw it back as such:

```
public RuntimeException launderThrowable(Throwable t) {
    if (t instanceof RuntimeException)
        return (RuntimeException) t;
    else if (t instanceof Error)
        throw (Error) t;
    else
        // this should never happen, since t was assumed unchecked
        throw new IllegalStateException("Not unchecked!", t);
}
```
Runnable, Callable, Future and FutureTask

```
Runnable
  run()

Future
  get(): V throws InterruptedException, ExecutionException, CancellationException
  cancel(mayInterruptIfRunning: boolean): boolean
  isDone(): boolean
  isCancelled(): boolean
  get(timeout: long, unit: TimeUnit): V throws InterruptedException/Execution/Timeout/CancellationException

RunnableFuture

FutureTask
  FutureTask(callable: Callable<V>)

Callable
  call(): V throws Exception
```
Consider the specification of a function that transforms an $A$ into a $V$ and that might be arbitrarily expensive:

```java
interface Computable<A, V> {
    V compute(A arg) throws InterruptedException;
}
```

We want to define a decorator that adds a layer of caching to the process:

```java
computable2 = new Memoizer(computable1)
```

The function `computable2` is equivalent to `computable1`, but caches repeated applications.
@ThreadSafe
public class Memoizer1<A, V> implements Computable<A, V> {
    @GuardedBy("this") private final Map<A, V> cache = new HashMap<>();
    private final Computable<A, V> c;

    public Memoizer1(Computable<A, V> c) {
        this.c = c;
    }

    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;
    }
}

Only a computation can run at a time, in sequence!
@ThreadSafe

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

The same computation might be executed more times!
@ThreadSafe
public class Memoizer3<A, V> implements Computable<A, V> {
    private final Map<A, Future<V>> cache = new ConcurrentHashMap<>();
    private final Computable<A, V> c;

    public Memoizer3(Computable<A, V> c) { this.c = c; }

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

Very rarely, the same computation might be executed more times!
Attempt #4: Exploit putIfAbsent of Concurrent Maps

@ThreadSafe
public class Memoizer<A, V> implements Computable<A, V> {
    private final Map<A, Future<V>> cache = new ConcurrentHashMap<>();
    private final Computable<A, V> c;
    public Memoizer(Computable<A, V> c) { this.c = c; }

    public V compute(A arg) throws InterruptedException {
        while (true) {
            Future<V> f = cache.get(arg);
            if (f == null) {
                FutureTask<V> ft = new FutureTask<>(() -> c.compute(arg));
                f = cache.putIfAbsent(arg, ft);
                if (f == null) {
                    f = ft; ft.run();
                }
            }
            try {
                return f.get();
            } catch (CancellationException e) {
                cache.remove(arg, f); // cache eviction, while (true) will retry
            } catch (ExecutionException e) {
                throw launderThrowable(e.getCause());
            }
        }
    }
}
Use the Concurrent Cache in the Factorization Servlet

```java
@ThreadSafe
@WebServlet("/Factorizer")
public class Factorizer extends StatelessFactorizer {
    private static final long serialVersionUID = 1L;
    private final Computable<BigInteger, BigInteger[]> cache
        = new Memoizer<>(arg -> factor(arg));

    @Override
    protected void doPost(HttpServletRequest request, HttpServletResponse response)
        throws IOException {
        BigInteger number = extractFromRequest(request);
        try {
            encodeIntoResponse(response, cache.compute(number));
        } catch (InterruptedException e) {
            encodeErrorIntoResponse(response, "factorization interrupted");
        }
    }
}
```
Exercise 1

Consider class MD5s, that implements a producer/consumer algorithm for generating the MD5 digests of files in some directories

1. Complete method index of the Indexer, in order to store in memory and print on standard output the MD5 digest of each file. The digest can be computed through the auxiliary md5 method.

2. Complete methods alreadyIndexed and getMD5sOf.

3. Complete method stop in order to interrupt all Indexers. Pending work should be processed, hence implement the handler for InterruptedException inside the run method of the Indexer.

If everything works, it should be possible to invoke MD5s from the command line and print the MD5s of the specified directories:

```
java -classpath examples/bin it.univr.concurrency.MD5s examples servlets
```

The execution should terminate at the end of the computation of the digests, always with the same printout (but typically in different orders).
Exercise 2

Implement a board for the CellularAutomata example, in order to implement the Game of Life. How do you implement the blocking waitForConvergence() method?