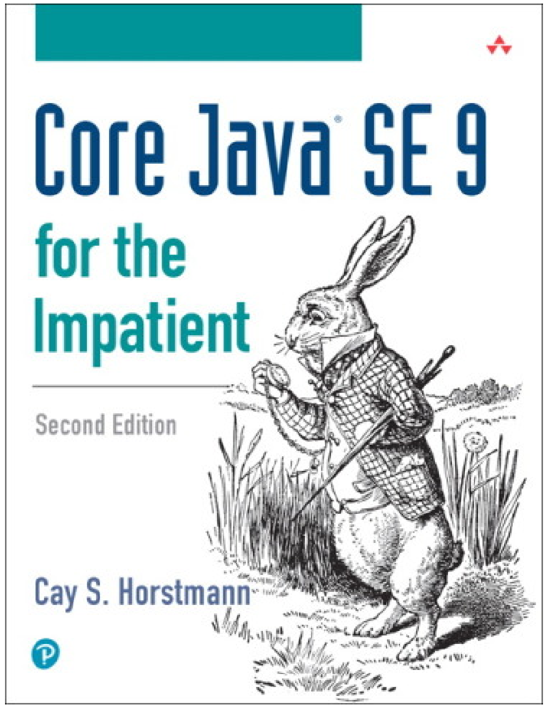
Java Concurrency For Humans

- Cay Horstmann
- Author of Core Java (10 editions since 1996)



Outline

- Audience: Application programmers
- Goal: Modern Concurrency Constructs
- Executors and Futures
- Asynchronous Processing with `CompletableFuture`
- Parallel Streams
- Safe Handoff
- Threadsafe Data Structures
  - And how to use them safely
- Cancelation
- What Not to Do

Old School Concurrency

- Given a `Set<Path>` of file paths and a string `word`, find all files that contain the word.
- Make a thread for each task.
- Use a lock around the result collection.

-
  ```
  Set<Path> results = new HashSet<>();
  for (Path p : paths)
     new Thread(() -> {
        if (word occurs in p)
           synchronized (results) { results.add(p); }
     }).start();
  ```
- That could be a lot of threads.
  - Thread creation and context switches are not free.
- Are you sure the hash set won't be corrupted?
- When is it all done?

Tasks, Not Threads

- Use an *executor* to execute tasks.
- A task can be a `Runnable` (presumably with a side effect):

```
Runnable task = () -> { ... };
ExecutorService exec = ...;
exec.execute(task);
```

- Or better, make the task compute a result:

```
Callable<Long> task = () -> { ...; return count; }
Future<Long> result = exec.submit(task);
```

- The result is a *future*—an object that represents a computation whose result will be available at some future time.

To Do:
[ ] Eat
X sleep
[ ] Knock stuff over
[ ] Run around like crazy at midnight
Found at funcatpictures.com

Executor Services

- The `Executors` has factory methods for making executor services:

```
ExecutorService exec = Executors.newCachedThreadPool();
    // Good for many tasks that are short-lived or mostly block
int processors = Runtime.getRuntime().availableProcessors();
int nthreads = processors - 2;
ExecutorService exec = Executors.newFixedThreadPool(nthreads);
    // Good for computationally intensive tasks
```

- The `ForkJoinPool` has a set of tasks queues (typically one per processor)
  - Idle processors “steal” tasks from busy ones
  - Good for workloads that recursively divide tasks into smaller ones
  - Bad for blocking tasks
- Java EE has `ManagedExecutorService`, `ScheduledExecutorService`
  - Access to contextual services
  - Transactions, security, etc.

Futures

- Submitting a `Callable` to an `Executor` yields a `Future`:

```
Callable<Long> task = () -> { ...; return count; }
Future<Long> resultFuture = exec.submit(task);
```

- A call to `get` blocks until the result is available:

```
Long actualResult = resultFuture.get();
```

- Block until all tasks are done:

```
Set<Path> paths = ...;
List<Callable<Long>> tasks = new ArrayList<>();
for (Path p : paths) tasks.add(() -> { ...; return count; });
List<Future<Long>> results = executor.invokeAll(tasks);
for (Future<Long> result : results) sum += result.get();
```

InvokeAny

- When searching for a match, want to stop after the first result becomes available.
- Use the `invokeAny` method:

```
List<Callable<Path>> tasks = new ArrayList<>();
for (Path p : files) tasks.add(
    () -> { if (word occurs in p) return p; else throw ... });
Path found = executor.invokeAny(tasks);
```

- As soon as a result is found, the other tasks are canceled.

Callable Demo

Asynchronous Processing

- When a thread waits for a result, it can't do work.
- Asynch I/O avoids blocking, using callbacks when results are available.
- Example: Play web framework—a few non-blocking threads serve many users.
- Requires asynchronous programming style.
- `CompletionStage<T>` interface provides many methods for composing callbacks.
- A `CompletableFuture<T>` is a `Future<T>` and a `CompletionStage<T>`

Working with Completable Futures

- Turn your processing pipeline into a sequence of methods. When a method is time-consuming, make it return a `CompletableFuture`:

```
public CompletableFuture<String> readPage(URL url)
public List<URL> getImageURLs(String webpage) // Not time-consuming
public CompletableFuture<List<BufferedImage>> getImages(List<URL> urls)
public void saveImages(List<BufferedImage> images)
```

- Now you can compose the operations:

```
CompletableFuture.completedFuture(urlToProcess)
    .thenComposeAsync(this::readPage, executor)
    .thenApply(this::getImageURLs)
    .thenCompose(this::getImages)
    .thenAccept(this::saveImages);
```

- All *xxx*`Async` methods optionally take an `Executor` argument

Dealing with Errors

- When any of the steps in the pipeline throws an exception, processing terminates with a `CompletionException` that wraps the original exception.
- You can substitute a value for an exception:

```
CompletableFuture.completedFuture(urlToProcess)
    .thenComposeAsync(this::readPage, executor)
    .exceptionally(ex -> "<html></html>")
    .thenApply(this::getImageURLs)
```

- Timeout handling:

```
CompletableFuture.completedFuture(urlToProcess)
    .thenComposeAsync(this::readPage, executor)
    .completeOnTimeout("<html></html>", 30, TimeUnit.SECONDS)
    .thenApply(this::getImageURLs)
```

- Or throw an exception instead:

```
    ...orTimeout(30, TimeUnit.SECONDS)
```

Combining Results

- Run two computations in parallel and combine results:

```
CompletableFuture<T> future1 = ...;
CompletableFuture<U> future2 = ...;
CompletableFuture<V> combined = future1.thenCombine(future2, combiner);
   // combiner takes arguments of type T and U, producing a result of type V
```

- `CompletableFuture.allOf` waits for multiple completable futures to complete, but it doesn't combine the results.
- If you are happy with either of two results, use the `applyToEither` method:

```
CompletableFuture<U> combined = future1.applyToEither(future2, transform);
    // transform maps T to U
```

- `CompletableFuture.anyOf` yields one result for a sequence of futures.
- In both cases, no cancellation of the other future(s)

Cancelation

- Java uses cooperative *interruption* mechanism.
- Cancelable task must periodically yield or monitor “interrupted” flag of the thread.
- `Future<T>` interface has a `cancel` method
- Canceling a `Future` produced by `ExecutorService.submit/invokeAny/invokeAll` works as expected.
- Canceling a `CompletableFuture` does not interrupt the underlying thread (because it has no idea what that might be)
- Various third party library implementations of “completable tasks” that are bound to an executor, such as https://github.com/vsilaev/tascalate-concurrent

CompletableFuture Demo

Parallel Streams

- Use parallel streams if you work on *in-memory* data and do *substantial work*:

```
long result = coll.parallelStream()
    .filter(e -> workHardToCheckSomeCondition(e)).count();
```

- The data structure needs to be splittable
  - Streams generate by `iterate` aren't splittable
  - `Files.lines` ok in Java 9, not ok in Java 8
- Of course, your lambdas need to be threadsafe

```
coll.parallelStream().forEach(s -> if (...) counter++; );
   // NO!!!—Use filter(...).count()
```

- Blocking in your lambdas might starve the fork-join pool
- It is possible to supply own executor:

```
ForkJoinPool executor = new ForkJoinPool(4);
ComputableFuture<Long> result = CompletableFuture.supplyAsync(() ->
   coll.parallelStream().filter(...).count(), executor);
```

Parallel Streams Demo

Concurrency–What Could Possibly Go Wrong?

- Concurrent programming is incredibly hard.
- Shared data can be corrupted.
- Program deadlocks when no thread can proceed.
- Bugs are nondeterministic.
  - “But it works on my machine!”
- Step 1: Understand what can go wrong.
- Step 2: Understand what you can do to avoid problems.

Visibility

- Two threads accessing the same variable:

```
private static boolean done = false;
Runnable hellos = () -> {
    doWork();
    done = true;
};
Callable goodbye = () -> {
    while (!done) sleep(1000);
    return doMoreWork(); // May never happen!
};
```

- The effect of `done = true;` in one thread is not *visible* to the other thread!
- Lack of visibility can be caused by caching.
  - RAM is slow, so each processor caches recently accessed variables.
- Lack of visibility can be caused by instruction reordering.

```
while (!done) i++;   ⇒ if (!done) while (true) i++;
```

Race Conditions

- Concurrent tasks update a shared counter:

```
private static volatile int count = 0;
...
count++; // Task 1
...
count++; // Task 2
...
```

- The update `count++` is not *atomic*.

```
register = count;
Increment register // What if context switch happens here?
count = register;
```

- It's not just counters:

```
// Add value to linked list queue
Node n = new Node();
if (head == null) head = n;
else tail.next = n;
tail = n; // What if context switch happens here?
tail.value = newValue;
```

Strategies for Safe Concurrency

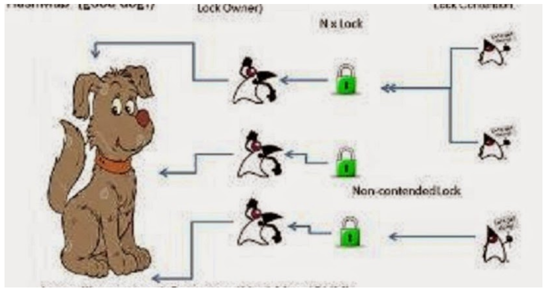
- Unfortunately, no equivalent to garbage collection for safe concurrency.
- Strategy: *confinement*.
  - Don't share data among tasks.
  - Example: Each task has its own result list, and the lists are combined after the tasks finish.
- Strategy: *immutability*.
  - It is safe to share immutable data structures.
  - Need special data structures for efficient accumulation.
- Strategy: *locking*.
  - Temporarily block other tasks when carrying out updates.
  - Can be expensive—other tasks wait idly.
  - Can be dangerous—deadlocks and subtle programming errors.
  - Best left to experts.

Confinement

- Example: Word count in multiple files.
- Updating a shared map is hard.
- Have each task compute a separate map.
- Safe handover to combining task:
  - Have each task *return* a map: `Callable<Map<String, Long>>`
  - Or put results into blocking queue.
- The data structure is never accessed by more than one task.

Concurrent Hash Maps

- The `java.util.concurrent` package supplies `ConcurrentHashMap` and other concurrent data structures.
- Safe to mutate concurrently.
- Clever implementations allow simultaneous updates in different parts of the hash table.
  - Don't try implementing this at home!
- Iterators are “weaky consistent”.
  - Elements present at the onset of the iteration are presented.
  - Later modifications may or may not be reflected.
  - No `ConcurrentModificationException`



Working with Concurrent Hash Maps

- `ConcurrentHashMap` won't be *damaged* by concurrent mutations.
- That doesn't mean that your algorithms are threadsafe:

```
Long oldValue = map.get(word);
Long newValue = oldValue == null ? 1 : oldValue + 1;
map.put(word, newValue); // NO!!!—might not replace oldValue
```

- Use methods for atomic updates:

```
map.compute(word, (k, v) -> v == null ? 1 : v + 1);
    // Or simply map.merge(word, 1L, Long::sum);
```

- Lambdas should complete quickly and not mutate the map!
- `computeIfPresent`, `computeIfAbsent`, `putIfAbsent`
- Bulk operations `addAll`, `forEach`, `reduce`, `search`, `replaceAll`

ConcurrentHashMap Demo

Immutable Classes

- Class is immutable if instance can't change after construction.
- Examples: `String`, `java.time.ZonedDateTime`
- But how do you collect results?.
- Using `HashSet` for aggregating results is dangerous:

```
results.add(newResult); // What if another thread accesses results?
```

- With a “persistent” set (not in the Java API), you can update like this:

```
results2 = results.add(newResult);
```

- Inexpensive—`results` and `results2` share most structure.
- Check out PCollections, Vavr, Cyclops, Paguro
- Not a silver bullet:

```
results = results.add(newResult); // Still a mutation
```

What About …

- Atomics?
- Intrinsic locks?

```
synchronized (hashTable) {
    for (K key : hashTable.keySet()) ...
}
```

```
synchronized ("LOCK") { // ?
    for (K key : hashTable.keySet()) ...
}
```

- Synchronized methods, `wait`, `notify`, `notifyAll`?
  - Per Brinch Hansen: “It is astounding to me that Java’s insecure parallelism is taken seriously by the programming community, a quarter of a century after the invention of monitors and Concurrent Pascal. It has no merit.” [Java’s Insecure Parallelism, ACM SIGPLAN Notices 34:38–45, April 1999.]
- `Semaphore`, `CountDownLatch`, `CyclicBarrier`, `Phaser`?

A Glimpse into the Future

- Concurrent programming model is awkward
- Mismatch between OS threads and tasks pushes APIs towards async
- Other languages have syntactic sugar for continuation passing:

```
async function getStuff() {
  const a = await getFirst();
  const b = await getSecond();
  return combine(a, b);
}
```

- A future version of Java may get “fibers”
  - Lightweight threads, like the “green threads” from Java 1.0
  - Blocking operation “parks” the fiber—very inexpensive
- May also get continuations:

```
generator() {
   while (...) {
      n = next(n);
      yield(n);
   }
}
```

```
main() {
   c = continuation(generator);
   x = c.continue();
   y = c.continue();
}
```

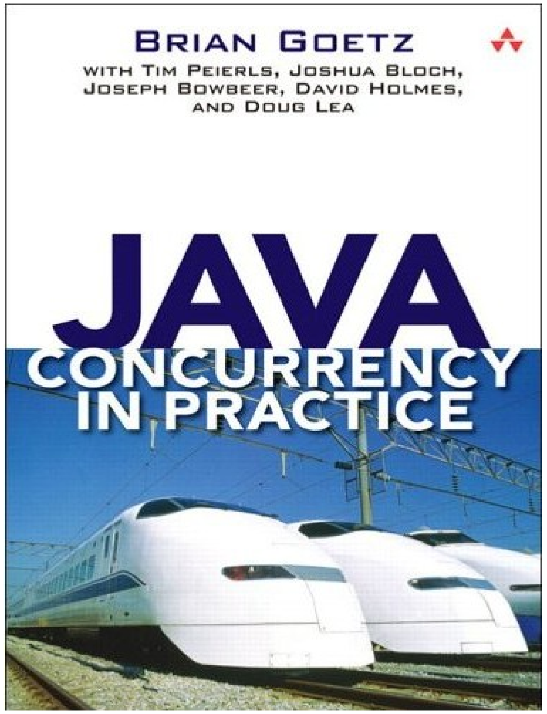
Summary

- Think tasks, not threads
- Pick the right executor service
- Use completable futures for asynchronous computation
- API has some rough edges—third party libraries may help
- Use parallel streams when appropriate (large in-memory data)
- Confinement, immutability, threadsafe data structures
- Don't use exotic stuff (phasers, cyclic barriers)
- Don't use what they taught you in school (locks and conditions)

CAN YOU TURN YOUR ONE-PAGE REPORT INTO A TWO-PAGE EXECUTIVE SUMMARY?

Where to Learn More

- Java Concurrency in Practice
- Core Java for the Impatient
- The Art of Multiprocessor Programming
- Tomasz Nurkiewicz' blog http://www.nurkiewicz.com
- Heinz Kabutz' newsletter http://www.javaspecialists.eu
- IBM Developerworks JVM Concurrency series
- Doug Schmidt's LiveLessons


Brian Goetz
with Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, and Doug Lea
JAVA CONCURRENCY IN PRACTICE