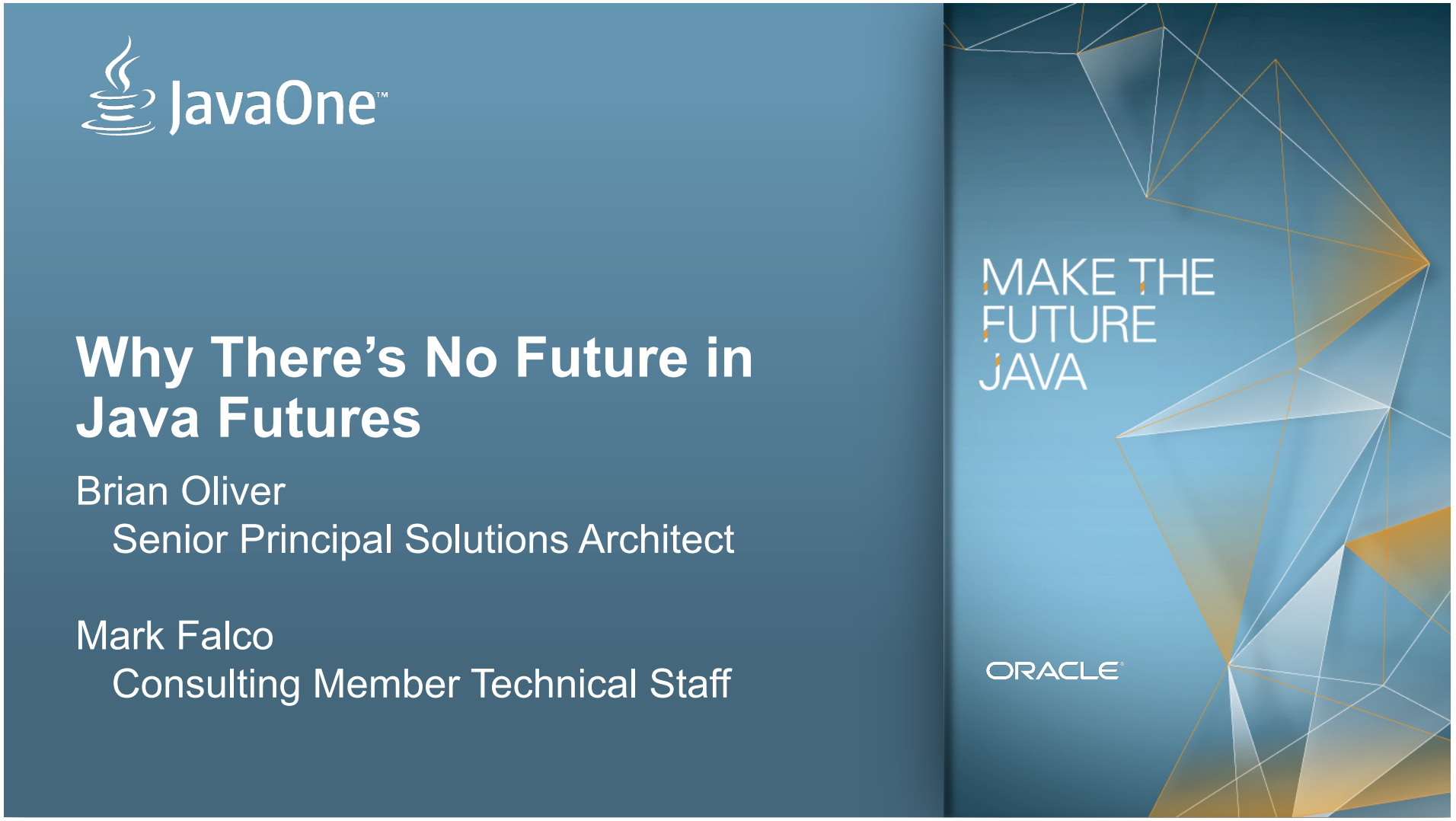

JavaOne™

The following is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions.
The development, release, and timing of any features or functionality described for Oracle's products remain at the sole discretion of Oracle.


JavaOne™
Why There's No Future in Java Futures
Brian Oliver
Senior Principal Solutions Architect
Mark Falco
Consulting Member Technical Staff
MAKE THE FUTURE JAVA
ORACLE

Introduction to Java Futures

Clarification…

- This talk is not about…
  - The future of Java
  - Future predictions about the future of Java
- It's about concurrent software development with Java
  - ie: the java.util.concurrent package

Introduction to Java Futures

java.util.concurrent.Future (from the Java Documentation)

- "A Future represents the result of an asynchronous computation."
- "Methods are provided to check if the computation is complete, to wait for its completion, and to retrieve the result of the computation."
- "The result can only be retrieved using the method 'get' when the computation has completed, **blocking** if necessary until it is ready."

Introduction to Java Futures

java.util.concurrent.Future (from the Java Documentation)

Type	Method & Description
V	**get**() *Waits if necessary for the computation to complete*, and then retrieves its result.*
V	**get**(long timeout, TimeUnit unit) *Waits if necessary* for at most the given time for the computation to complete, and then retrieves its result, if available.*
boolean	**cancel**(boolean mayInterruptIfRunning) *Attempts to cancel execution of this task.*
boolean	**isCancelled**() *Returns true if this task was cancelled before it completed normally.*
boolean	**isDone**() *Returns true if this task completed.*

** unless an exception is thrown*

Introduction to Java Futures

java.util.concurrent.Future

- Consider the following:

```
/**
 * Provides a mechanism to search an archive.
 */
interface ArchiveSearcher {
   /**
    * Search an archive for some artifact, returning the result location.
    */
   String search(String artifact);
}
```


Introduction to Java Futures

Asynchronous Search Example

```
void showSearch(final String artifact,
                ExecutorService executor,
                final ArchiveSearcher searcher) throws InterruptedException {

   Future<String> future = executor.submit(new Callable<String>() {
      public String call() {
         return searcher.search(artifact);
      }});

   displayOtherThings(); // do other things while searching

   try {
      displayText(future.get());       // use future
   } catch (ExecutionException ex) {
      cleanup();
   }
}
```


Introduction to the Java ExecutorService

java.util.concurrent.ExecutorService

- Most instances of Java Futures come from Java ExecutorServices
- “An Executor … provides methods to manage termination and methods that can produce a Future for tracking progress of one or more asynchronous tasks.”
- And by “tracking progress” we mean isDone, isCancelled (not percentage complete)

Introduction to the Java ExecutorService

java.util.concurrent.ExecutorService (selected methods)

Type	Method & Description
Future<T>	**submit(Callable<T> task)** *Submits a value-returning task for execution and returns a Future representing the pending results of the task.*
Future<?>	**submit(Runnable task)** *Submits a Runnable task for execution and returns a Future representing that task.*
<T> List<Future<T>>	**invokeAll(Collection<? extends Callable<T>> tasks)** *Executes the given tasks, returning a list of Futures holding their status and results when all complete.*
<T> T	**invokeAny(Collection<? extends Callable<T>> tasks)** *Executes the given tasks, returning the result of one that has completed successfully (i.e., without throwing an exception), if any do.*

Styles of adoption – Blocking

Blocking wastes a thread ☹

```
Future<Double> f = executor.submit(task);

// do something else for a while (or nothing at all)

Double d = f.get(); // block!
```


Styles of adoption - Polling

Polling wastes a core ☹

```
Future<Double> f = executor.submit(task);

while (!f.isDone()) { // poll
    // do something else for awhile (or nothing)
    }

Double d = f.get());
```

ExecutorService… is just gets worse

Go parallel and block ☹

```
List<Future<Double>> futures = service.invokeAll(tasks);
double total = 0;
for (Future f : futures) {
    total += f.get();  //block, block, block …
    }
double avg = total / tasks.size();
proceed(avg);
```


Typical Requirements For “going async”.

Let’s step back a minute… what do you really do?

- Perform a task in the background and we either…
  a. Don’t care if or when it completes
  b. Do care if and/or when it completes, in which case…
     i. We do something with the result (or exception)
     ii. We don’t care about the result (or exception)

     In Summary:
     We want to do something after a task has completed, perhaps with the result… ie: A Continuation!

- This is not exactly what Futures or ExecutorServices provide

Introduction to Continuations

(Wikipedia)

- “a continuation is an abstract representation of the control state of a computer program”
- Useful for representing “control mechanisms in programming languages such as exceptions, generators, co-routines, and so on.”
- ie: Continuations are a way to represent “do this after completion of a task”.

What's Missing?

A callback from the Future…

- ExecutorService methods should take a "callback" or an "observer" to notify upon completion of a task
  - Allows us to know about completion (when it occurs)
  - Allows us to proceed immediately with other work (not wait)
  - Removes the need for blocking for results
  - Removes the need for polling of results
  - Allows for continuation style processing (good for Java 8)

Updating Async APIs for callbacks

- Asynchronous methods should **always** return **void** but take an ObservableFuture

```
Future<T> executor.submit(Callable<T> task);
```

thus becomes

```
<T> void executor.invoke(Callable<T> task, ObservableFuture<T>);
```

- When a task has completed, the executor “notifies” the supplied ObservableFuture.

Introducing ObservableFutures

A simple solution

```
interface ObservableFuture<T> extends Future<T> {
    void onResult(T);
    void onFailure(Exception);
    }
```


Using Observable Futures

Improved Asynchronous Search Example

```
void showSearch(final String artifact,
                ExecutorService executor,
                final ArchiveSearcher searcher) throws InterruptedException {

   ObservableFuture<String> future = new ObservableFuture<String>() {
      public void onResult(String result) { displayTest(result); }
      public void onFailure(ExecutionException e) { cleanup(); }
   };

   executor.submit(new Callable<String>() {
      public String call() {
         return searcher.search(artifact);
      }}, future);

   displayOtherThings(); // do other things while searching
}
```


Using Async APIs with callbacks

- It is what happens after invocation that matters…
  - Previously we were doing a:

```
future.get();
… now we do …
return;
```

- With ObservableFutures, we're now **non-blocking and non-polling!**

Continuations are key

- Rather than block and wait for a result, when one is provided to you, you process or act upon it (ie: Event Driven!)
- ObservableFuture becomes a Continuation

AggregatingFuture

Last Future result proceeds with calculation

```
class AggregatingFuture<T> impl ObservableFuture<T> {
    void onResult(T result) {
        synchronized (sharedResults) {
            sharedResults.add(result);
            if (sharedResults.size() == cRequired) {
                proceed(aggregate(sharedResults));
            }
        }
    }
}
```


The big problem with ObservableFutures!

- In our example we didn't implement all of the Future methods! Oops! Where's cancel(), isCancelled(), get(), isDone()…

```
interface ObservableFuture<T> extends Future<T> { … }
```

```
ObservableFuture<String> future = new ObservableFuture<String>() {
  public void onResult(String result) { displayTest(result); }
  public void onFailure(ExecutionException e) { cleanup(); }
};
```

- Do we actually need the other methods?

Introducing Collectors

A more flexible option

```
interface Collector<T> {
    void add(T);
    void flush()  default { };
}
```


Collector

Basics

Type	Method & Description
<T> void	**add**(T result) *Called by asynchronous implementations to provide (potentially a partial) result.*
void	**flush**() *Called by asynchronous implementations to signal it's time to process previously added results.*

- This combination permits map reduce style processing
  - (if that's what you're into)

Using Collectors

Improved Asynchronous Search Example

```
void showSearch(final String artifact,
                ExecutorService executor,
                ArchiveSearcher searcher) throws InterruptedException {

   Collector<String> collector = new Collector<String>() {
      public void add(String result) { displayTest(result); }
   };

   executor.submit(new Callable<String>() {
      public String call() {
         return searcher.search(artifact);
      }}, collector);

   displayOtherThings(); // do other things while searching
}
```


Supporting Exceptions?

Ok, this is getting a bit harder

- How does an implementation “notify” a Collector about an Exception?
- Poor
  - `Collector<?> + instanceof Exception`
- Better
  - `Collector.onFailure(ExecutionException e)`
- Best

```
interface Result<T> {
   T get() throws ExecutionException;   //does not block!
}
…
<T> void submit(Task, Collector<Result<T>>);
```


Using Result Collectors for Exceptions

```
void showSearch(final String artifact,
                ExecutorService executor,
                final ArchiveSearcher searcher) throws InterruptedException {

   Collector<Result<String>> collector = new Collector<Result<String>>() {
      public void add(Result<String> result) {
         try {
            displayTest(result.get());
         } catch (ExecutionException e) { cleanup(); }
      }
   };

   executor.submit(new Callable<String>() {
      public String call() {
         return searcher.search(artifact);
      }}, collector);

   displayOtherThings(); // do other things while searching
}
```


Supporting Cancellation / Interruption?

Please stop!!!

- Not all types of work can or should be cancelled / interrupted
  - Futures actually allow this! ☹
- How to cancel / interrupt?
  - Should the “request” to interrupt be on the “task” or on the “result”?
- How to know if a task was cancelled / interrupted?
  - Represented via `CancelledException` from `Result.get()`
  - Represented via `InterruptedException` from `Result.get()`
- `Future.cancel()` is optional!

Introducing Interruptable

Use a “wrapper” to support cancellation and interruption

```
interface Interruptable {
   void interrupt();
   boolean isInterrupted();
}

class InterruptableCallable<T> implements Callable<T>, Interruptable {
…
}

Class InterruptableRunnable implements Runnable, Interruptable {
…
}
```


Changes to Result

Interruption indicated via Exceptions

```
interface Result<T> {
   T get() throws ExecutionException,
                  CancelledException,
                  InterruptedException
}
```

```
Interruptable task = new InterruptableCallable(callable);
executor.submit(task, collector);

// do stuff… oops must interrupt!
task.interrupt();
```


Supporting Timeouts?

You only have so much time to do this…

- A task must execute with in a certain period of time, after which you give up!

- Similar solution to Interruptable… but add parameter to executor.
  - Represented via `TimedOutExecption` from `Result.get()`

```
void submit(Callable<T>, Collector<T>, int timeout, TimeUnit timeoutUnit);


interface Result<T> {
   T get() throws ExecutionException, InterruptedException
                  CancelledException, TimedOutException
}
```


Supporting Progress Feedback

How much work has been done?

- Two forms of progress?
  - Declared progress from serial set of tasks.
  - Inferred progress from multiple parallel tasks

- One solution?
  - Represent Progress as a Result with more “fidelity”

```
interface Progress<T> extends Result<T> {
   int getPercentageComplete();
   long getTimeRemaining(TimeUnit unit);
}
```


Replace Callables and Runnables with Tasks

Cleaning up the API… do we really need Callables/Runnables?

- Introduce Task
  - Provides a Collector to the Task
  - The Task can then provide Results (and Progress) or flush()
  - Provide wrapper and/or support for Callables and Runnables;

```
/**
 * An task to be performed asynchronously, the result of
 * which is placed in the provided Collector.
 */
interface Task<T> {
   void execute(Collector<? extends Result<T>> collector);
}
```


Using Collectors with Multiple Results

Go parallel and continue

```
abstract class AggregatingCollector<X, Y> implements Collector<X> {
   private List<X> results;
   . . .
   void add(X result) {
      results.add(result);
   }

   void flush() {
      proceed(aggregate(results));
   }

   abstract Y aggregate(List<X> results);

   abstract void proceed(Y result);
}
```


FutureCollector

Futures “we’re not dead yet”… You can have it both ways… ☺

```
class FutureCollector<T> implements Future<T>, Collector<Result<T>> {
   private T result = null;

   void synchronized add(Result<T> result) {
      this.result = result;
      notifyAll();
   }

   T synchronized get() throws … {
      while (result == null) {
         wait();
      }
      return result.get();
   }
   …
}
```


The CollectorExecutorService

Perhaps it should be like this?

Type	Method & Description
void	**submit(Task<T> task, Collector<? Extends Result<T>> collector)** *Submits a task for execution, the result of which is added to the provided collector. A null collector indicates no result is required.*
void	**submit(Task<T> task, Collector<? Extends Result<T>> collector, long timeout, TimeUnit timeUnits)** *Submits a task for execution, the result of which is added to the provided collector. A null collector indicates no result is required. The result must be provided with in the specified time, otherwise a TimedOutException will be added as the result.*
void	**invokeAll(Collection<? extends Task<T>>, Collector<? extends Result<T>>)** *Executes the given tasks, added each result into the provided collector. A null collector indicates no result is required.*

Questions

Sessions of Interest

Session	Day/Time	Location
Distributed Caching to Data Grids: The Past, Present and Future of Scalable Java	Monday 3:00 – 4:00	Parc 55 Market Street
Sharding Middleware to Achieve Elasticity and High Availability in the Cloud	Wednesday 1:00 – 2:00	Parc 55 Market Street
Using the New javax.cache Caching Standard	Thursday 11:00 – 12:00	Parc 55 Cyril Magnin 1
NoSQL Usage Patterns in Java Enterprise Applications	Thursday 3:30 – 4:30	Parc 55 Mission

Join the Coherence Community

http://coherence.oracle.com


@OracleCoherence

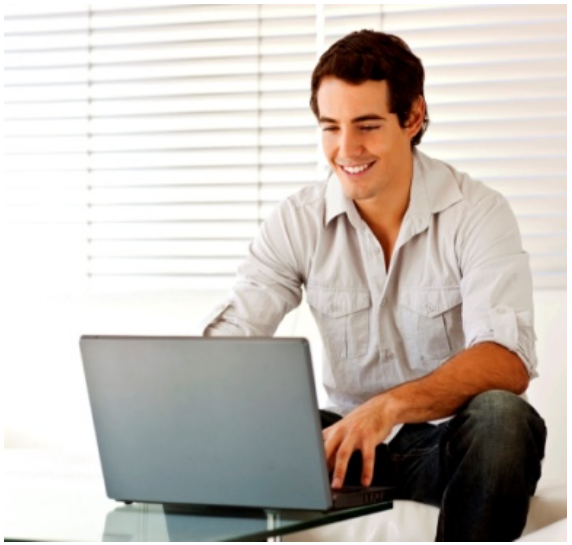

facebook.com/OracleCoherence


blogs.oracle.com/OracleCoherence


Oracle Coherence Users


youtube.com/OracleCoherence


coherence.oracle.com/display/CSIG
Coherence Special Interest Group



JavaOne ORACLE


JavaOne™