The Past, Present, and CompletableFuture

...

Rob Signorelli
VP Product Development - LifeShare Technologies

Ye Olde Agenda Slide

- What is the deal with the asynchronous craze?
- Some core “functional” terminology so we speak the same language
  - apply, accept, and lambdas
- What is a Java “Future” and why it needed a reboot
- Java 8’s CompletableFuture
  - get(), thenApply(), thenAccept(), thenCompose()
- Dealing with multiple parallel futures
  - thenCombine(), anyOf(), allOf()
- Where did that thread come from?
- A sample API/app that puts all of this to use

Why Asynchronous Is So Popular Right Now

- Anecdotally, too many developers flock to it without knowing why they do
  - ... but the craze is not without merit
- When combined with sound functional principles it can result in clear, well-abstracted code
  - A chain of easily readable methods as opposed to jumping from class to class
  - Better type safety and null safety
- When used with async I/O you can release a thread until the work is complete
  - Anywhere from 3-8% of CPU time can be spent context switching (worst case)
  - Fewer threads means each server can handle more work (devops thanks you)

Functional Concepts: Function<T, R> (apply/map)

Take a value and apply a transformation that results in some other value

```
Function<String, String> upper = new Function<>() {
    public String apply(String input) {
        return input.toUpperCase();
    }
};

// words == [“FOO”, “BAR”, “BAZ”]
List<String> words = Arrays.asList(“foo”, “bar”, “baz”)
    .stream()
    .map(upper)
    .collect(Collectors.toList());
```

Functional Concepts: Function<T, R> (apply/map)

Even cooler with lambdas!

```
// words == [“FOO”, “BAR”, “BAZ”]
List<String> words = Arrays.asList(“foo”, “bar”, “baz”)
    .stream()
    .map(w -> w.toUpperCase())
    .collect(Collectors.toList());

// Also supports “method reference” notation
.map(String::toUpperCase)
```

Functional Concepts: Consumer<T> (accept)

Take a value and do some “terminal” work with it

```
Consumer<String> printMe = new Consumer<>() {
    public void accept(String input) {
        System.out.println(input);
    }
};

// Prints each word on separate lines
Arrays.asList(“foo”, “bar”, “baz”)
    .stream()
    .forEach(printMe);
```

Functional Concepts: Consumer<T> (accept)

Again, lambdas rock!

```
// Prints each word on separate lines
Arrays.asList(“foo”, “bar”, “baz”)
    .stream()
    .forEach(w -> System.out.println(w));

// Can use method references here, too
.forEach(System.out::println);
```

Functional Concepts: Supplier<T> (supply/get)

From an anonymous context (i.e. no args) produce/supply a value.

```
Supplier<String> nobody = new Supplier<>() {
    public String get() {
        return “nobody”;
    }
};

// First one is “Bob” and second one is “nobody”
String a = Optional.ofNullable(“Bob”).orElseGet(nobody);
String b = Optional.ofNullable(null).orElseGet(nobody);
```

Functional Concepts: Supplier<T> (supply/get)

Again, lambdas rock!

```
// No need to define separate or anonymous class
String a = Optional.ofNullable(“Bob”).orElseGet(() -> “nobody”);

// Any zero-arg function that returns correct type works
public String defaultPerson() { return “nobody”; }
…
String a = Optional.ofNullable(“Bob”).orElseGet(this::defaultPerson);
```

The Original Java Future

- Been around since Java 5
- Abstraction for an asynchronous process that will eventually result in a value in the... FUTURE!
- Functions that return futures (theoretically) return immediately
  - Value has NOT been generated yet (most likely)
  - Control goes back to original code as quickly as possible
- Receiver of the future decides at what point blocking is necessary to wait for value

FutureExample1.java
(the “hello world” of Futures)

FutureExample2.java
(the Future is messy)

Shortcomings of Future

- Must to come from Executor - can't just "make one"
  - Not without writing your own Future implementation, anyway
- If you have one, you have no way to just mark it as complete
- Need to pass Futures instead of values to subsequent steps that use the value
  - Not a clean API
  - Can call .get() right before next step in the process but that defeats the purpose
- Not great when you have to wait on multiple futures
  - Can wait for each one individually like the example, but stuck waiting on slowest one
  - Generic implementation for N is pretty verbose using standard JDK code
- What if you need to do more work with the resulting value?
  - Code typically gets ugly with less obvious "flow"
  - Results in more thread dispatching than necessary

Java 8's CompletableFuture

- Still represents some process that generates a value in the... FUTURE!
- Defines a "pipeline" of tasks that make up a much larger operation
  - Similar to Promise from JS
  - Think typical web app flow: auth, validate input, query, business logic, serialize results, and respond
- CompletableFuture is the reference implementation of CompletionStage
- Don't be afraid of the near 70-method API for the class
  - Most are slight variants of the same handful of operations
  - Yes, some of the methods are atrociously named...
- Alternate style to RxJava - not better, not worse - just different

CompletableFutureExample1.java
(better than the original)

CompletableFutureChaining.java
(code that reads like a fine novel)

Dealing With Multiple CompletableFutures

- What if an operation in the chain returns another CompletionStage?
  - Use thenApply() when operation returns the exact value to pass along.
  - Use thenCompose() when operation returns another CompletableFuture with the desired value.
- What if you have multiple futures and you need to wait on all of them to complete?
  - Use the static CompletableFuture.allOf() method.
  - Returns a single future w/ a ‘Void’ type so you only know when they’re done.
  - If you want the values from each future, we’ve got a workaround.
- What if you have multiple futures and you just want the first one that finishes?
  - Use the static CompletableFuture.anyOf() method.
  - Returns a single future w/ the same generic type as the original futures.
  - Subsequent completions are ignored.
  - No clue why they made the generic type Object instead of inferring ‘T’

CompletableFutureCombine.java
(flatten those futures)

CompletableFutureMultiple.java
(wait for any/all operations to complete)

CompletableFutureExample2.java
(much better than the original)

What Thread Am I Running On?

- Pipeline executes on the thread that called `future.complete(VALUE)`
- Use the thenXyzAsync() variants of methods to control thread flow.
- Uses ForkJoinPool.commonPool() if no executor service is specified.
- Be careful when program is dependent on ThreadLocal data!!!

```
// Thread 1 is used to build ‘future’
future = CompletableFuture
    .supplyAsync(this::doStuffA)
    .thenApply(this::doStuffB)
    .thenApplyAsync(this::doStuffC);
```

```
public String doStuffA() {
    // This is run in Thread 2
}
public String doStuffB() {
    // This is run in Thread 2
}
public String doStuffC() {
    // This is run in Thread 3
}
```

CompletableFutureThreads.java (threads make my head hurt)

Api.java
(a “real world” example of CF in action)

An painfully basic order management system w/ inventory and order processing.

Additional Links/Resources

- Tomasz Nurkiewicz talk on CompletableFuture - best one out there - https://vimeo.com/131394616
- Same dude's website/blog which is all about Java development - http://www.nurkiewicz.com/
- Javadoc for CompletableFuture