Scaling Java Applications Through Concurrency

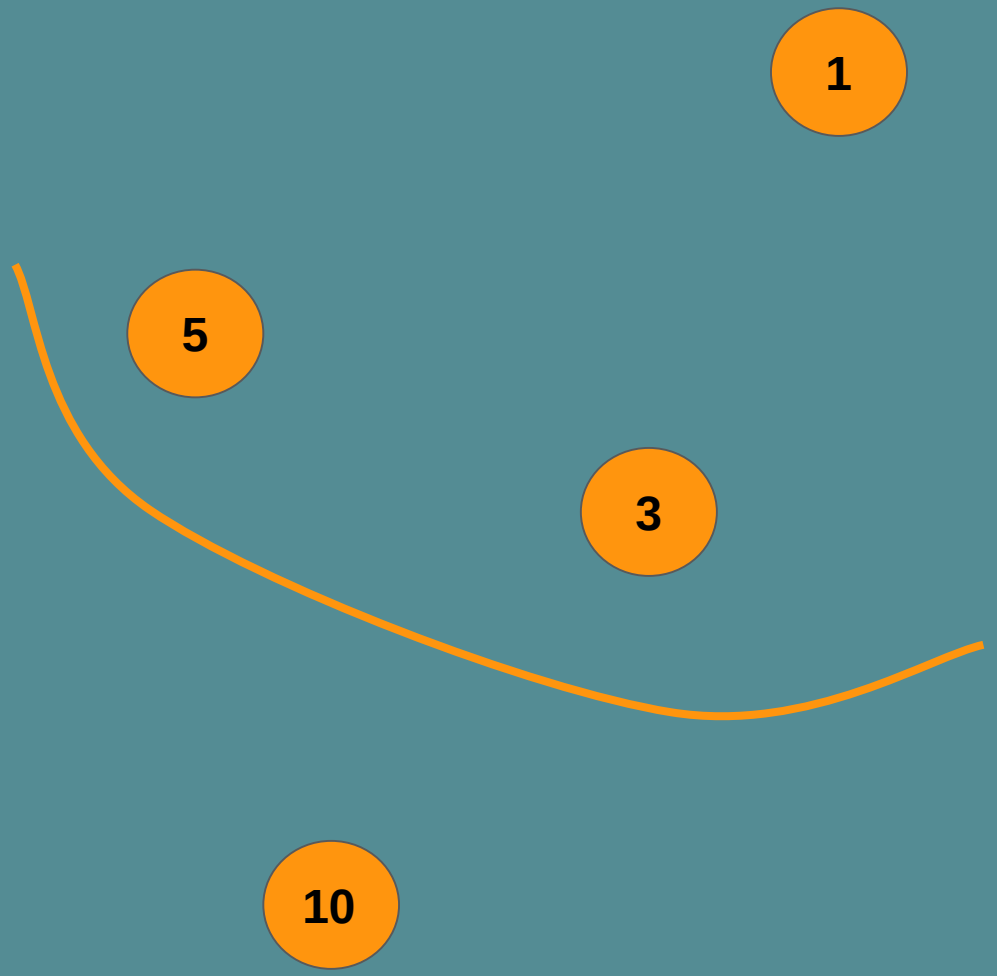
Coordinating Efforts Among Dependent Processes

Josh Cummings

@jzheaux | tech.joshcummings.com

Identity Pipeline: Dependencies

- **Reads in a stream of identity-related data**
- **Normalizes and verifies the contents of each identity**
- Merges and persists the identities in an in-memory cache
- Notifies an error queue if any of the above fails
- Records aggregate identity statistics



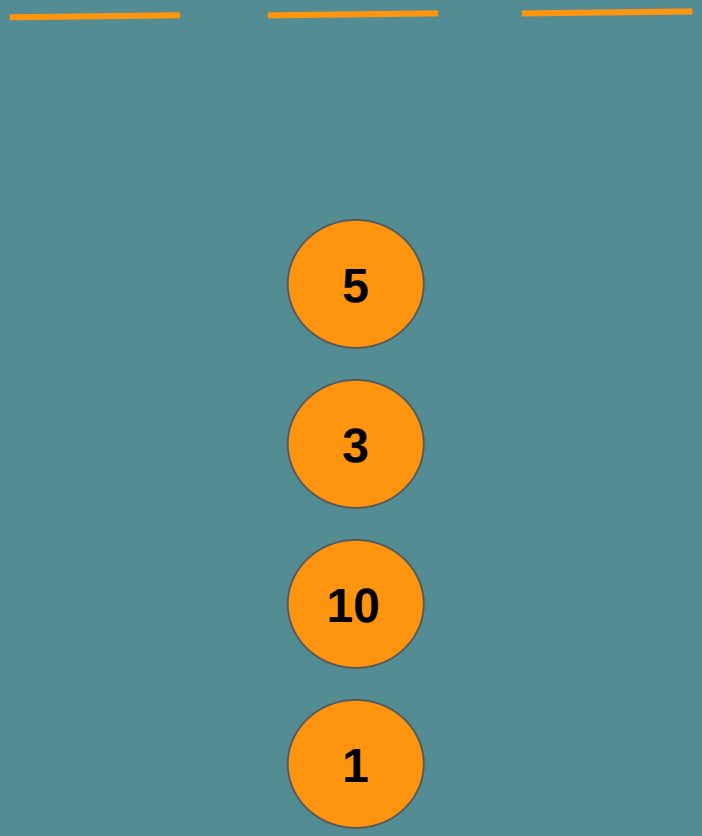

```
// In this case, order doesn't really matter;
// however it is implicit. verifyEmailAddress
// cannot be called until verifyPhoneNumber
// is finished


boolean pVerified =
verifyPhoneNumber(identity);
boolean eVerified =
verifyEmailAddress(identity);
boolean aVerified = verifyAddresses(identity);


// A simple way to multi-thread this is to
// introduce multiple workers


pool.submit(() -> verifyPhoneNumber(identity));
pool.submit(() -> verifyEmailAddress(identity));
pool.submit(() -> verifyAddresses(identity));
```

// How to make sure nothing happens until all

CountDownLatch

- Similar to, but richer semantics than **join**
- Threads wait for a programmatically decremented count down before continuing
- Helpful when one thread is dependent on the completion or arrival of several other threads before moving on

```
// guests arrive on their own and the waiter
// waits for everyone to arrive

CountDownLatch orchestrator = new
CountDownLatch(numberOfActors);


public void perform(Actor actor) {
        waiter.countDown(); // sing song,
etc...
}

public void sendNextActor()
                throws InterruptedException
{
        orchestrator.await(); // send now...
}
```




```
// In this case, order doesn't really matter;
// however it is implicit. verifyEmailAddress
// cannot be called until verifyPhoneNumber
// is finished


boolean pVerified =
verifyPhoneNumber(identity);
boolean eVerified =
verifyEmailAddress(identity);
boolean aVerified = verifyAddresses(identity);


// A simple way to multi-thread this is to
// introduce multiple workers


pool.submit(() -> verifyPhoneNumber(identity));
pool.submit(() -> verifyEmailAddress(identity));
pool.submit(() -> verifyAddresses(identity));
```

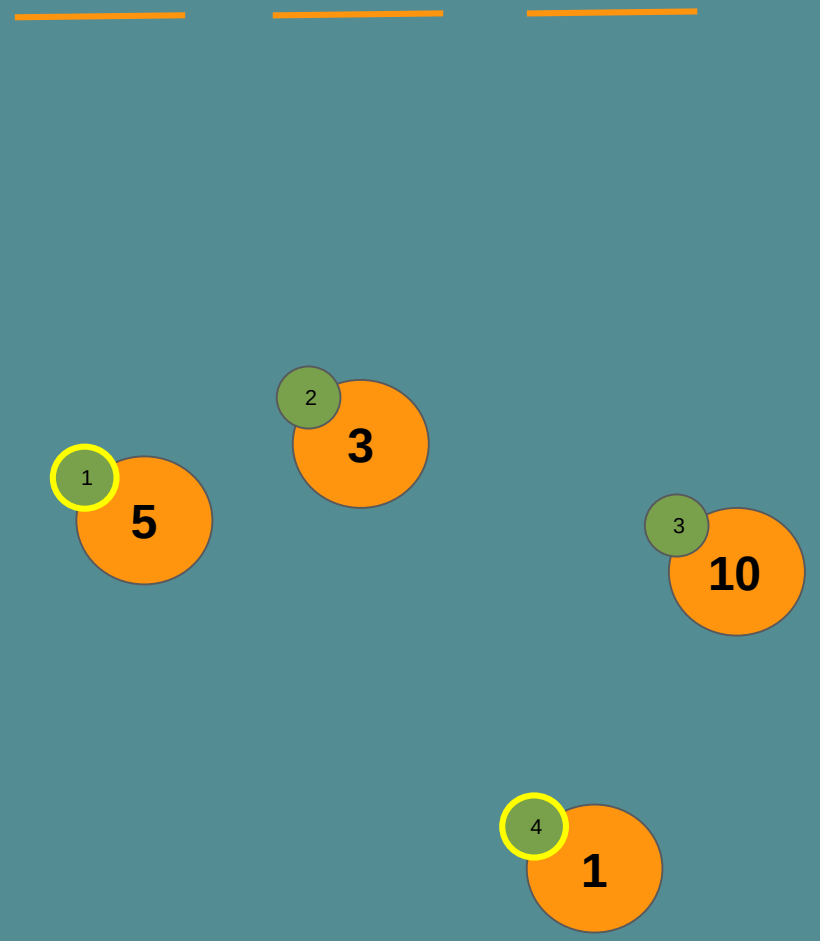
Why stay in line?

Benefits of promises

Order represented, but not rigid

Tickets can be obtained by one customer and then handed to another customer

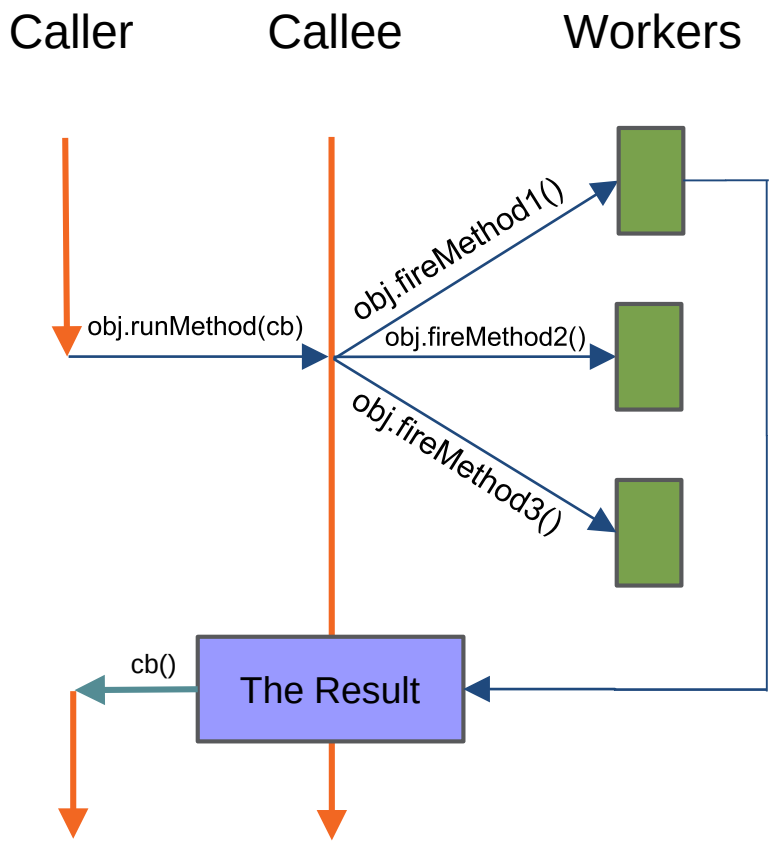
Customers can do something else while they wait (e.g. leave the waiting room and come back when it's their turn)




```
// Futures afford the abstraction of a callback
// instead in a return value


Future<Boolean> pResult = pool.submit(() ->
        verifyPhoneNumber(identity));
Future<Boolean> eResult = pool.submit(() ->
        verifyEmailAddress(identity));
Future<Boolean> aResult = pool.submit(() ->
        dresses(identity));

// A promise affords the possibility of waiting
// on results independent of execution order


if ( eResult.get() && aResult.get() &&
pResult.get() ) {
        // do something amazing here
}
```

The order of execution is independent from the completion order

Continuation-Passing Pattern



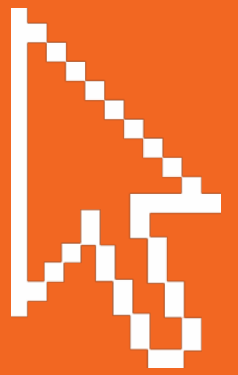
- Caller invokes method, providing a callback method
- Caller does not need to block on thread completion
- Callee can easily make the choice, even at runtime, to execute caller's request concurrently

Method References & Lambdas

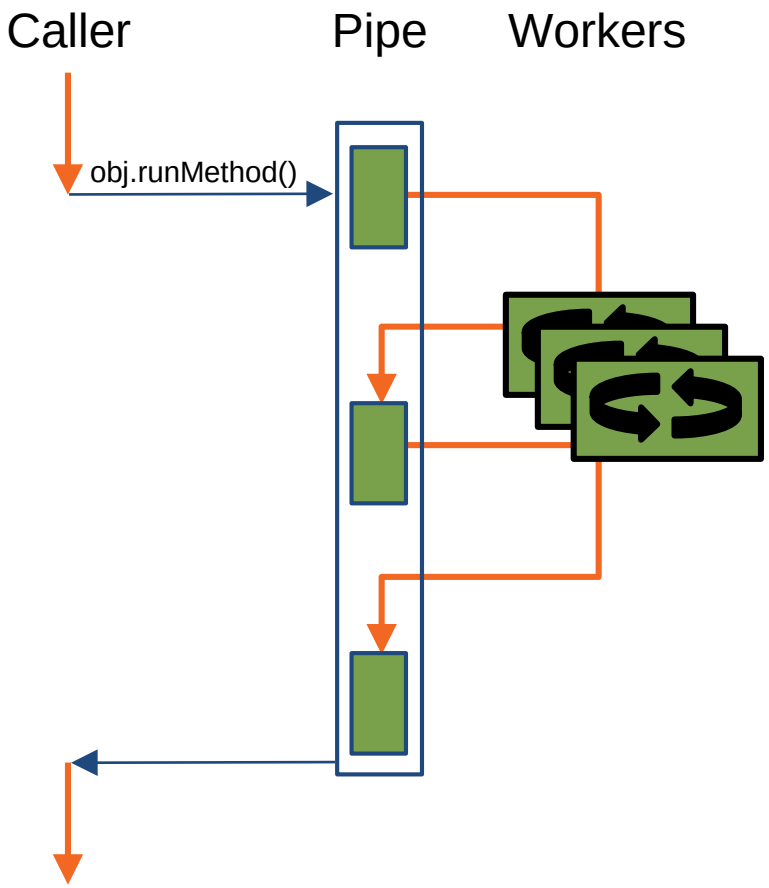
- Simple way to specify a callback to a downstream execution
- The caller can now do other things at the same time while waiting for the code to finish
- Passing more than one callback allows the caller to choreograph execution as more of a state transition diagram

```
// allow the callee to notify the caller when it
// is complete by using a callback

format(identity, this::fail, this::persist);

void fail(Identity identity, Exception e) {
  malformed.addIdentity(identity, e);
}


void persist(Identity identity) {
  doPersist(identity, this::fail, this::doStats);
}
```

Continuations typically return immediately, but that doesn't mean the work is done



Pipeline Pattern



- Flattens out otherwise deeply-nested callbacks inherent in the Continuation-Passing pattern
- Abstracts away dependencies between asynchronous tasks
- Affords the same benefits of ignoring, delegating, or blocking on the result as Futures

CompletableFuture

- Fluent API allows for chain of responsibility, similar to the Java Stream API
- Uses the ForkJoin common pool, though each method takes an **ExecutorService** as a parameter for easy configuration.
- Semantics similar to **Future**; fire-and-forget, block, or delegate the result

```
// run one task after another completes
// asynchronously

CompletableFuture c =
    CompletableFuture.runAsync(
            () -> str.toUpperCase())
            .thenAcceptAsync(
                    (upperCased) ->

            upperCased.replace("a", "b"), pool)
            .exceptionally(
                    (e) ->
                            log.error(e)
            );
```

Asynchronous Composition

Pattern	… using CompletableFuture
Fire-and-Forget	.runAsync
Foot-Race	.anyOf → .thenAcceptAsync
Scatter-Gather	.allOf → .thenAcceptAsync

Review

- Dependencies can be managed either from an orchestration or choreography standpoint
- CountDownLatch, Continuation-Passing, and CompletableFuture are all ways to think choreographically about concurrent dependencies
- Consider partitioning thread pools to increase throughput