Asynchronous

Monday, April 9, 2018 10:31 AM

Synchronicity

- Call method -> result of method return should be available
- Code only continues executing after last line
- Asynchronous call = execution continues immediately, code left to run (forking)

`Thread.sleep(time)`

- InterruptedException thrown when Ctrl-C
- Catch exception, task.cancel()

Future<V>

- Represents result of asynchronous task
- Implemented by RecursiveTask, RecursiveAction

`get()` returns the result of the computation (waiting for it if needed).

`get(timeout, unit)` returns the result of the computation (waiting for up to the timeout period if needed).

`cancel(interrupt)` tries to cancel the task -- if `interrupt` is true, cancel even if the task has started. Otherwise, cancel only if the task is still waiting to get started.

`isCancelled()` returns `true` of the task has been cancelled.

`isDone()` returns `true` if the task has been completed.

CompletableFuture<V>

- Specify a callback to be executed when asynchronous task is complete
  - Instead of busy waiting - unnecessarily checks if code is complete
- Implements Future, CompletionStage
- Can pass in completed value (see below - completedFuture) / asynchronous task to be executed

```
CompletableFuture
    .supplyAsync(() -> m1.multiply(m2))
    .thenAccept(System.out::println);
```

`.join()`

- Blocks code until CompletableFuture completes

`.allOf(), .anyOf()`

- Waits for tasks to complete when combined with .join()

Functor

- thenApply takes a Function, returns a CompletableFuture

Other Methods

`thenRun`, which takes a `Runnable`,

`thenAcceptBoth`, which takes a `BiConsumer` and another `CompletableFuture`

`thenCombine`, which takes a `BiFunction` and another `CompletableFuture`

`thenCompose`, which takes in a `Function` `fn`, which instead of returning a "plain" type, `fn` returns a `CompletableFuture`.

- thenRun, thenAccept = terminal, thenApply = intermediate

Monad

- `thenCompose`
  - Takes in a function that returns a monad, and returns a monad
- `completedFuture`
  - Wraps value with a CompletableFuture, similar to .of()

Variations

- Either, Both, Executor (running in default ForkJoinPool is not enough), Throwable

Handling Exceptions

- Where should the exception be handled?
- `isCompletedAbnormally(), getException()`

`.whenComplete()`

- Terminal operation with BiConsumer(result, exception), similar to peek()
- handle() takes in a BiFunction

`.exceptionally()`

- Replaces thrown exception with a value (like orElse in Optional)

Exercises

1.
```
D d = CompletableFuture.supplyAsync(() -> f(a))
          .thenApply(g)
          .thenApply(h)
          .get();
```
2.
```
CompletableFuture.supplyAsync(() -> f())
          .thenApply(g)
          .thenAccept(h);
```
3.
```
CompletableFuture.supplyAsync(() -> f(a))
          .thenApply(result -> {
              CompletableFuture<C> c = CompletableFuture.supplyAsync(() ->
      g(result));
              CompletableFuture<D> d = CompletableFuture.supplyAsync(() ->
      h(result));
              return c.thenCombine(d, i);
               });
```