CS2030 Lecture 11

Asynchronous Programming

Henry Chia (hchia@comp.nus.edu.sg)

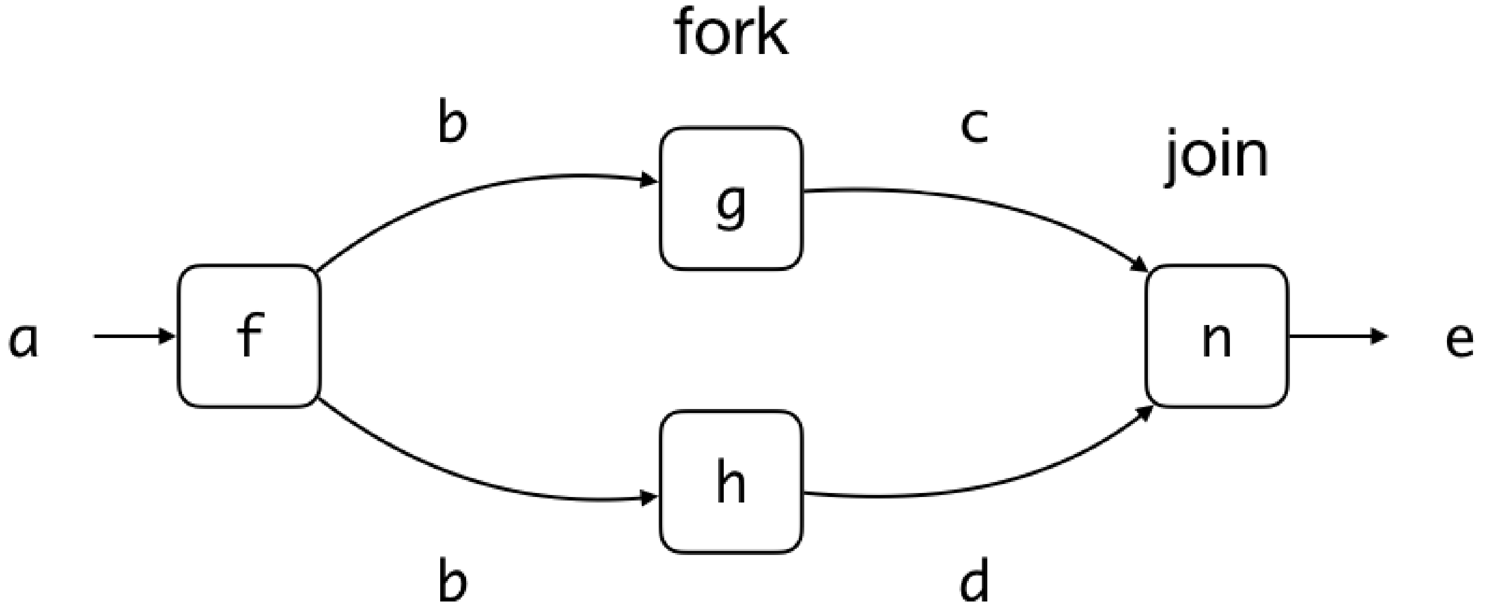
Semester 1 2023 / 2024

Lecture Outline

- □ Able to identify fork and join processes from a given computation graph
- □ Understand the difference between synchronous and asynchronous programming
- □ Appreciate asynchronous programming in the context of spawning threads to perform tasks
- □ Able to define asynchronous computations via Java's `CompletableFuture`
- □ Use of a callback to execute a block of code when an asynchronous task completes
- □ Able to convert synchronous code to an asynchronous version

Fork and Join

```
b = f(a);
c = g(b);
d = h(b);
e = n(c,d);
```



- `f` invoked before `g` and `h`; `n` invoked after `g` and `h`
- If `g` and `h` does not produce side effects (i.e. does not depend or change external states), then
  - **fork** task `g` to execute at the same time as `h`, then
  - **join** back task `g` later

Synchronous Programming

```java
int doWork(int n) {
    try {
        Thread.sleep(n * 1000); // Thread.sleep throws InterruptedException
    } catch (InterruptedException e) { }
    return n;
}
```

```java
B f(A a) {
    System.out.println("f: start");
    doWork(a.x);
    System.out.println("f: done");
    return new B();
}

C g(B b, int n) {
    System.out.println("g: start");
    doWork(n);
    System.out.println("g: done");
    return new C();
}
```

```java
D h(B b, int n) {
    System.out.println("h: start");
    doWork(n);
    System.out.println("h: done");
    return new D();
}

E n(C c, D d) {
    System.out.println("n: proceeds");
    return new E();
}
```

Synchronous Programming

- Synchronous programming: one function executes at a time

```
jshell> void foo(int m, int n) {
   ...>     B b = f(new A(5));
   ...>     C c = g(b, m);
   ...>     D d = h(b, n);
   ...>     E e = n(c, d);
   ...> }
|  created method foo(int,int)

jshell> foo(5, 10)
f: start // f starts @ t = 0s
f: done  // f completes @ t = 5s
g: start // g starts after f completes @ t = 5s
g: done  // g completes after five seconds @ t = 10s
h: start // h starts after g completes @ t = 10s
h: done  // h completes after another 10 seconds @ t = 20s
n: proceeds // n proceeds @ t = 20s
```

- Since the execution of `g` and `h` can start at the same time
  - should require only 10 seconds to complete the execution of both methods, i.e. total time is $5 + 10 = 15$ seconds

Asynchronous Programming with Threads

- Spawn a separate process thread to compute `g`

```
jshell> void foo(int m, int n) throws InterruptedException {
   ...>     B b = f(new A(5));
   ...>     Thread t = new Thread(() -> g(b, m));
   ...>     t.start();
   ...>     h(b, n);
   ...>     t.join(); // join() throws InterruptedException
   ...>     System.out.println("n: proceeds");
   ...> }
|  created method foo(int,int)
```

- A `Runnable` is passed to the `Thread` constructor
  - `Runnable` has the single abstract method `void run()`

```
jshell> Runnable r =
   ..>>    () -> System.out.println("hello");
r ==> $Lambda$...

jshell> r.run()
hello
```

```
jshell> r = () -> 1
r ==> $Lambda$...

jshell> r.run()
```

Thread Completion via **join()**

- Wait for thread to complete using the `join()` method
  - `join()` method is blocking, i.e. blocks the thread and returns only when execution of the thread completes

```
jshell> foo(5, 10) // completes after 15 seconds
f: start
f: done
h: start
g: start
g: done
h: done // t.join() returns immediately as g has already completed
n: proceeds
```

```
jshell> foo(10, 5) // completes after 15 seconds
f: start
f: done
h: start
g: start
h: done
g: done // t.join() waits another 5 seconds for g to complete
n: proceeds
```

Java's `CompletableFuture<T>` as a Promise

- Computation context that handles asynchronous computations and *promises* completion, and possibly returning a value
  - static methods:
    - `supplyAsync` that takes in `Supplier`
    - `runAsync` that takes in `Runnable`
  - encapsulates the thread which starts execution rightaway
- Result of asynchronous computation is obtained via `join()`

```
jshell> CompletableFuture<B> cf = CompletableFuture.
   ...>    supplyAsync(() -> f(new A(5)))
f: start
cf ==> java.util.concurrent.CompletableFuture@4783da3f[Not completed]

jshell> // do other work

jshell> cf.join() // waits for thread to complete
f: done
$.. ==> B@37a71e93
```

Passing Callbacks into **`CompletableFuture`**

- Suppose a `CompletableFuture` is currently running a task
  - a *callback* is any executable code that is passed to the `CompletableFuture`, so that this code can be *called after* the current task completes
- *Hollywood Principle*: “Don’t call us, we’ll call you (back)”
- Example: callback passed via `thenApply` (like a `map`!)

```
jshell> CompletableFuture<C> cf = CompletableFuture.
   ...>    supplyAsync(() -> f(new A(5))).
   ...>    thenApply(x -> g(x, 5))
cf ==> java.util.concurrent.CompletableFuture@6193b845[Not completed]
f: start

jshell> cf.join()
f: done
g: start
g: done
$.. ==> C@c4437c4
```

Callbacks in `CompletableStage`

- While `CompletableFuture` provides the static constructors, `CompletionStage` provides methods that take in callbacks
  - `thenAccept(Consumer<? super T> action)`
  - `thenApply(Function<? super T, ? extends U> fn)`
  - `thenCompose(Function<? super T, ? extends CompletableStage<U>> fn)`
  - `thenCombine(CompletionStage<? extends U> other,`
    `BiFunction<? super T, ? super U, ? extends V> fn)`
- `thenApply` and `thenCompose` are analogous to `map` and `flatMap` in `Optional`, `Stream`, etc.
  - `CompletableFuture` is a Functor and a Monad
- `join()` waits for thread execution to complete and returns the result, or returns `Void` for `Runnable` tasks

Converting Synchronous to Asynchronous

- Given the following synchronous program fragment

```
int foo(int x) {
    if (x < 0) {
        return 0;
    } else {
        return doWork(x);
    }
}
```

- The asynchronous version is

```
CompletableFuture<Integer> fooAsync(int x) {
    if (x < 0) {
        return CompletableFuture.completedFuture(0);
    } else {
        return CompletableFuture.supplyAsync(() -> doWork(x));
    }
}
```

- `CompletableFuture.completedFuture(U value)` wraps a completed value in a `CompletableFuture`

Converting Synchronous to Asynchronous

- Suppose method `bar` is defined as

  ```
  int bar(int x) {
      return doWork(x);
  }
  ```

  with the sequence of synchronous method calls given as

  ```
  int y = foo(5)
  int z = bar(y)
  ```

  or

  ```
  int z = bar(foo(5))
  ```

- The equivalent asynchronous version is expressed as follows:

  ```
  jshell> CompletableFuture<Integer> cf = fooAsync(5).
     ...>     thenApply(x -> bar(x))
  cf ==> java.util.concurrent.CompletableFuture@50040f0c[Not completed]

  jshell> // do other work

  jshell> int z = cf.join()
  z ==> 5
  ```

Converting Synchronous to Asynchronous

- Now suppose `bar` is also asynchronous

```
CompletableFuture<Integer> barAsync(int x) {
    return CompletableFuture.supplyAsync(() -> doWork(x));
}
```

- Then the equivalent asynchronous version is expressed as:

```
jshell> cf = fooAsync(5).thenCompose(x -> barAsync(x)) // how about thenApply?
cf ==> java.util.concurrent.CompletableFuture@1b2c6ec2[Not completed]

jshell> int z = cf.join()
z ==> 5
```

- Combine results of `CompletableFutures` via a `BiFunction`

```
jshell> cf = fooAsync(5).thenCombine(barAsync(5), (x, y) -> x + y)
cf ==> java.util.concurrent.CompletableFuture@1e80bfe8[Not completed]

jshell> int z = cf.join()
z ==> 10
```

- Both `fooAsync` and `barAsync` must be completed, before resulting `CompletableFuture` from `thenCombine` completes

Async Variants of Callback Methods

- Callback methods have an `Async` variant

```
jshell> void foo() {
   ...>     CompletableFuture<Void> cf1 = CompletableFuture.runAsync(() -> {
   ...>         doWork(5);
   ...>         System.out.println("cf1: " + Thread.currentThread().getName()); });
   ...>
   ...>     CompletableFuture<Void> cf2 = cf1.thenRun(() -> {
   ...>         doWork(5);
   ...>         System.out.println("cf2: " + Thread.currentThread().getName()); });
   ...>
   ...>     CompletableFuture<Void> cf3 = cf1.thenRunAsync(() -> {
   ...>         doWork(5);
   ...>         System.out.println("cf3: " + Thread.currentThread().getName()); });
   ...>
   ...>     cf2.join();
   ...>     cf3.join();
   ...> }
|  created method foo()

jshell> foo()
cf1: ForkJoinPool.commonPool-worker-3
cf2: ForkJoinPool.commonPool-worker-3
cf3: ForkJoinPool.commonPool-worker-5
```

  - What if `thenRun` (or `thenRunAsync`) is used for both?
  - What if `thenRun` and `thenRunAsync` are switched?

CompletableFuture Asynchronous Computation

- Constructing the `CompletableFuture` pipeline out of given synchronous methods:

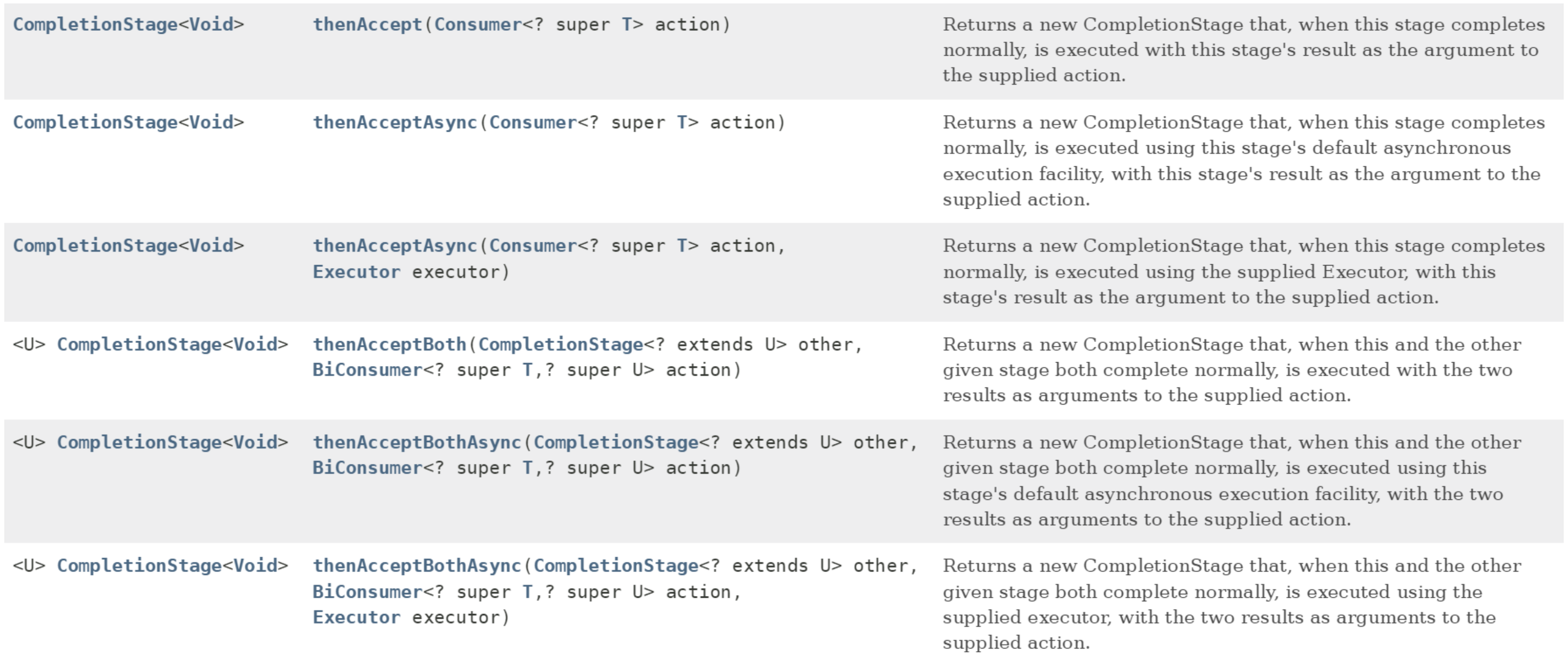
```java
E foo(int m, int n) {
    Supplier<B> suppB = () -> f(new A());

    CompletableFuture<B> cfB = CompletableFuture.supplyAsync(suppB);
    CompletableFuture<C> cfC = cfB.thenApply(x -> g(x, m));
    CompletableFuture<D> cfD = cfB.thenApplyAsync(x -> h(x, n));
    CompletableFuture<E> cfE = cfC.thenCombine(cfD, (c,d) -> n(c, d));

    E e = cfE.join();

    return e;
}
```

```
jshell> foo(5, 10)
f: start // t = 0s
f: done  // t = 5s
g: start // t = 5s
h: start // t = 5s
g: done  // t = 10s
h: done  // t = 15s
$.. ==> E@49097b5d
```

```
jshell> foo(10, 5)
f: start // t = 0s
f: done  // t = 5s
g: start // t = 5s
h: start // t = 5s
h: done  // t = 10s
g: done  // t = 15s
$.. ==> E@37a71e93
```

More Methods in `CompletionStage`

- Methods that take callbacks of the form `then<X><Y><Z>`
  - `X` is `Accept`, `Combine`, `Compose`, `Run`, …
  - `Y` is nothing, `Both`, `Either`
  - `Z` is nothing or `Async`



`CompletionStage<Void>`	`thenAccept(Consumer<? super T> action)`	Returns a new CompletionStage that, when this stage completes normally, is executed with this stage's result as the argument to the supplied action.
`CompletionStage<Void>`	`thenAcceptAsync(Consumer<? super T> action)`	Returns a new CompletionStage that, when this stage completes normally, is executed using this stage's default asynchronous execution facility, with this stage's result as the argument to the supplied action.
`CompletionStage<Void>`	`thenAcceptAsync(Consumer<? super T> action, Executor executor)`	Returns a new CompletionStage that, when this stage completes normally, is executed using the supplied Executor, with this stage's result as the argument to the supplied action.
`<U> CompletionStage<Void>`	`thenAcceptBoth(CompletionStage<? extends U> other, BiConsumer<? super T,? super U> action)`	Returns a new CompletionStage that, when this and the other given stage both complete normally, is executed with the two results as arguments to the supplied action.
`<U> CompletionStage<Void>`	`thenAcceptBothAsync(CompletionStage<? extends U> other, BiConsumer<? super T,? super U> action)`	Returns a new CompletionStage that, when this and the other given stage both complete normally, is executed using this stage's default asynchronous execution facility, with the two results as arguments to the supplied action.
`<U> CompletionStage<Void>`	`thenAcceptBothAsync(CompletionStage<? extends U> other, BiConsumer<? super T,? super U> action, Executor executor)`	Returns a new CompletionStage that, when this and the other given stage both complete normally, is executed using the supplied executor, with the two results as arguments to the supplied action.