# CS2030 Lecture 11

## Asynchronous Programming

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

Semester 2 2022 / 2023



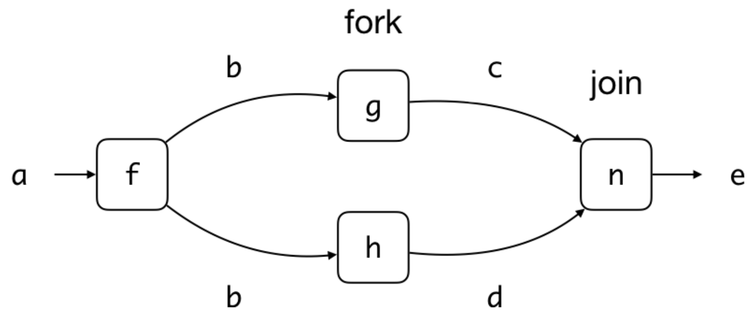
# Lecture Outline

- 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(a)` invoked before `g(b)` and `h(b)`; `n(c,d)` invoked after
- 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 sleep(int n) { // to simulate a heavy task
    try {
        Thread.sleep(n * 1000); // Thread.sleep throws InterruptedException
    } catch (InterruptedException e) { }
    return n;
}

B f(A a) {
    System.out.println("f: start");
    sleep(5);
    System.out.println("f: done");
    return new B();
}

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

D h(B b, int n) {
    System.out.println("h: start");
    sleep(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());
   ...>     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());
   ...>     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> void f() {}
|  created method f()

jshell> Runnable r = () -> f()
r ==> $Lambda$...

jshell> r.run()
```

```
jshell> int g() { return 1;}
|  created method g()

jshell> r = () -> g()
r ==> $Lambda$...

jshell> r.run()
```



## Thread Completion via `join()`

- Wait for thread to complete using the `join()` method
  - `join()` method is blocking and returns 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>`

- Static methods `supplyAsync` (and `runAsync`) creates instances of `CompletableFuture` out of `Supplier`s (and `Runnable`s)
  - returns a `CompletableFuture` rightaway
  - encapsulates the thread which starts execution

```
jshell> CompletableFuture.supplyAsync(() -> f(new A()))
$.. ==> java.util.concurrent.CompletableFuture@5d099f62[Not completed]
f: start

jshell> f: done
```

- Result of asynchronous computation is obtained via `join()`

```
jshell> CompletableFuture.supplyAsync(() -> f(new A())).join()
f: start
f: done
$.. ==> B@37bba400
```

# Callback

- A *callback* is any executable code that is passed as an argument to other code
  - so that the former can be called back (or called after) upon completion of the latter
- Based on the *Hollywood Principle*:

  *"Don't call us, we'll call you (back)"*

- A callback function is passed into `thenApply` (just a `map`!)

```
jshell> CompletableFuture.supplyAsync(() -> f(new A())).
   ...> thenApply(x -> g(x, 5)).
   ...> join()
f: start
f: done
g: start
g: done
$.. ==> C@31cefde0
```



# Callbacks in CompletableStage

- While `CompletableFuture` provides the static constructors, `CompletionStage` provides the other callback methods, e.g.
  - `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 as well as a Monad!
- The `join()` method is blocking and returns the result when execution completes; returns `Void` for `Runnable` tasks
- *Reminder: Always ensure that* `join()` *is called!*



# `CompletableFuture` Asynchronous Computation

- Constructing the `CompletableFuture` pipeline:

```
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
```



# Converting Synchronous to Asynchronous

- Give 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 we have the following synchronous method calls

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

with **bar** defined as

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

- The above sequence of function calls can be composed as

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

- The equivalent asynchronous version is

```
int z = fooAsync(5).
    thenApply(x -> bar(x)).
    join();
```



## Converting Synchronous to Asynchronous

- What if we switch the method calls, i.e.

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

and suppose **bar** is asynchronous as well, i.e.

```
CompletableFuture<Integer> barAsync(int x) {
    return CompletableFuture.completedFuture(x);
}
```

- Then the equivalent asynchronous version is

```
int z = barAsync(5).
    thenCompose(y -> fooAsync(y)).
    join();
```

- What if we use `thenApply` instead of `thenCompose`?



## Combining Completable Futures

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

```
int z = fooAsync(5).
    thenCombine(barAsync(5), (x,y) -> x + y).
    join()
```

- Both `fooAsync` and `barAsync` must be completed, before resulting `CompletableFuture` from `thenCombine` completes
- To summarize...
  - use `runAsync` or `supplyAsync` to create
  - then<X><Y><Z> where
    - X is `Accept`, `Combine`, `Compose`, `Run`, ...
    - Y is nothing, `Both`, `Either`
    - Z is nothing or `Async`



## Async Variants of Callback Methods

- Callback methods have an `Async` variant, e.g.
  - thenRun/thenRunAsync *may* run on a different thread
  - thenRun runs on the same thread if still busy

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