CS2030 Lecture 12

Asynchronous Programming

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

Semester 1 2020 / 2021

Lecture Outline

- Synchronous programming
- Asynchronous programming
  - Thread creation
  - Busy waiting
  - Thread completion
- Callback
- `CompletableFuture`
- Converting synchronous to asynchronous computations

Synchronous vs Asynchronous Programming

- The following task `Unit` task defined for instrumentation

```java
class UnitTask {
    int id;

    UnitTask (int id) {
        this.id = id;
    }

    int compute() {
        String name = Thread.currentThread().getName();
        try {
            System.out.println(name + " : start");
            Thread.sleep(id * 1000);
            System.out.println(name + " : end");
        } catch (InterruptedException e) { }
        return id;
    }
}
```

Synchronous Computation

- Typical program involving synchronous computations

```
public static void main(String[] args) {
    System.out.println("Before calling compute()");
    new UnitTask(Integer.valueOf(args[0])).compute();
    System.out.println("After calling compute()");
}
```

- When calling a method in synchronous programming, the method gets executed, and when the method returns, the result of the method (if any) becomes available
- Clearly, `compute()` method needs to return before `main` method resumes with the final output
- The method might delay the execution of subsequent methods

Asynchronous Computation

- Create a thread that runs the `compute` method

```
public static void main(String[] args) {
    System.out.println("Before calling compute()");
    Thread t = new Thread(
            () -> new UnitTask(Integer.valueOf(args[0])).compute()
    t.start();
    System.out.println("After calling compute()");
}
```

- Passing a `Runnable` to the `Thread` constructor
- `Runnable` is a SAM with the abstract `run()` method
- Start the thread with `start()` method to execute the `compute` on a separate thread
  - use `run()` method to execute on the same thread

Busy Waiting

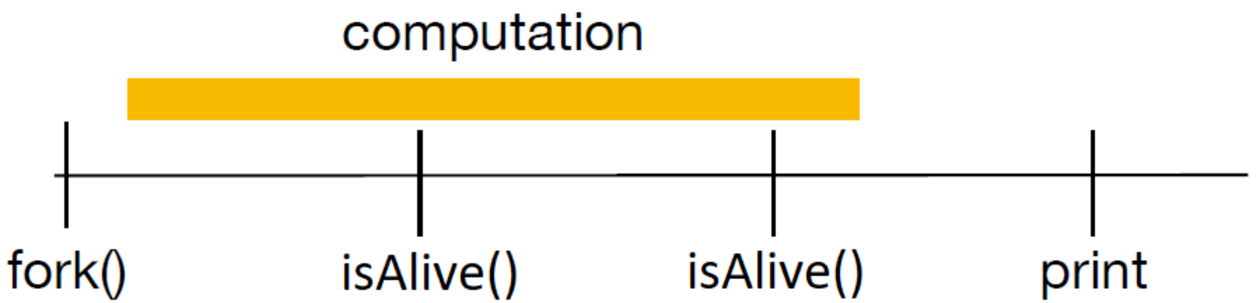
- Waiting for a thread to complete using a busy-waiting loop

```
System.out.println("Before calling compute()");
Thread t = new Thread(
        () -> new UnitTask(Integer.valueOf(args[0])).compute());
t.start();
System.out.println("After calling compute()");

while (t.isAlive()) {
    new UnitTask(Integer.valueOf(args[1])).compute();
    System.out.print(".");
}

System.out.println("compute() completes");
```

- `t.isAlive()` is called after every fixed time interval



Busy Waiting

- Performing an unrelated task while waiting

```
System.out.println("Before calling compute()");
Thread t = new Thread(
        () -> new UnitTask(Integer.valueOf(args[0])).compute());
t.start();
System.out.println("After calling compute()");

System.out.println("Do independent task...");
new UnitTask(5).compute();
System.out.println("Done independent task...");

while (t.isAlive()) {
    new UnitTask(Integer.valueOf(args[1])).compute();
    System.out.print(".");
}

System.out.println("compute() completes");
```

Thread Completion via **join()**

- Wait for thread to complete using the `join` method

```
try {
    System.out.println("Before calling compute()");

    Thread t = new Thread(
            () -> new UnitTask(Integer.valueOf(args[0])).compute()
    t.start();

    System.out.println("Do independent task...");
    new UnitTask(5).compute();
    System.out.println("Done independent task...");

    System.out.println("Waiting at join()");
    t.join();
    System.out.println("After calling compute()");

} catch (InterruptedException e) { }
```

- `join()` throws `InterruptedException` if the current thread is interrupted

Callback

- Rather than busy-waiting, a *callback* can also be specified
  - A callback (more aptly call-after) is any executable code that is passed as an argument to other code so that the former can be called back (executed) at a certain time
  - The execution may be immediate (synchronous callback) or happen later (asynchronous callback)
  - Avoid repetitive checking to see if the asynchronous task completes
  - Callback may be invoked from a thread but is not a requirement
  - An *observer pattern* can be utilized where the callback can be invoked, say `notifyListener`

Callback

- The *conventional* way of creating a listener is via an interface
- Motivated by the *Observer* pattern which addresses the issue of tight coupling using *Inversion of Control*

```java
interface Listener {
    public void notifyListener();
}
```

- Caller implements Listener with a `notifyListener()` method
- Listener(s) (or observers) are included in the thread
- Thread notifies the listener(s) when execution completes
- Tasks dependent on the completion of execution of the thread can be initiated as part of the notification

Callback via Listener

```java
class Async<T> {
    Thread thread;
    Listener listener;

    Async<T> doThis(Runnable r) {
        thread = new Thread(() -> r.run());
        thread.start();
        return this;
    }

    Async<T> thenDoThis(Listener listener) {
        this.listener = listener;
        return this;
    }

    void join() {
        try {
            this.thread.join();
        } catch (InterruptedException e) { }
        this.listener.notifyListener();
    }
}
```

Callback via Listener

- Client creates an asynchronous task to be spawned on a seperate thread

```
new Async<Integer>()
    .doThis(() ->
            new UnitTask(Integer.valueOf(args[0])).compute())
    .thenDoThis(() ->
            System.out.println("async task completes"));
    .join();

System.out.println("client continues...");
```

- In method `join()`, `this.thread.join()` waits for the asynchronous thread to complete
- What if `join()` was not invoked?
  - Client continues execution *immediately after* asynchronous task is spawned on another thread

Callback via Function

```java
class Async<T> {
    Thread thread;
    T value;
    Function<T,T> listener;

    Async<T> doThis(Supplier<T> s) {
        thread = new Thread(() -> {
            this.value = s.get();
        });
        this.thread.start();
        return this;
    }

    Async<T> thenDoThis(Function<T,T> listener) {
        this.listener = listener;
        return this;
    }

    T join() {
        try {
            this.thread.join();
            this.value = this.listener.apply(this.value);
        } catch (InterruptedException e) { }
        return value;
    }
}
```

Callback via Function

- While the thread is executing, client proceeds to execute independent tasks
- Dependent tasks are done after `join()` returns

```java
Async<Integer> async = new Async<Integer>();

async.doThis(() ->
        new UnitTask(Integer.valueOf(args[0])).compute())
    .thenDoThis(x -> x + 1);

System.out.println("client continues...");
new UnitTask(5).compute(); // independent task

int result = async.join();
System.out.println(result); // dependent task
```

- `Async` class is a *promise* that a value will be returned so that a value can be obtained, or a callback can be executed

Implementing Promise via `CompletableFuture`

- static methods `runAsync` and `supplyAsync` creates `CompletableFuture` instances of out `Runnable` and `Suppliers` respectively

```java
System.out.println("Before calling compute()");

CompletableFuture<Integer> cf = CompletableFuture
        .supplyAsync(() ->
                new UnitTask(Integer.valueOf(args[0])).compute());

System.out.println("Do independent task...");
new UnitTask(5).compute();
System.out.println("Done independent task...");

Integer result = cf.join();
System.out.println("After compute(): " + result);
```

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

Callbacks via Chaining `CompletionStages`

- `thenAccept()` accepts a `Consumer` and the future chain passes the result of computation to it
- Returns a `CompletableFuture<Void>`

```
System.out.println("Before calling compute()");

CompletableFuture<Void> cf = CompletableFuture
    .supplyAsync(() ->
            new UnitTask(Integer.valueOf(args[0])).compute())
    .thenAccept(s ->
            System.out.println("After compute(): " + s));

System.out.println("Do independent task");
new UnitTask(5).compute();
System.out.println("Done independent task");
cf.join();
```

- The `join()` method is blocking and returns the result when complete. What happens when `join` is not called?

Callbacks via Chaining `CompletionStages`

- While `CompletetableFuture` provides the static `CompletableFuture` constructors (`supplyAsync` and `runAsync`), `CompletionStage` provides the other callback methods
  - `thenAccept(Consumer<? super T>) action`
  - `thenApply(Function<? super T, ? extends U> fn`
  - `thenCombine(CompletionStage<? extends U> other, BiFunction<? super T, ? super U, ? extends V> fn)`
  - `thenCompose(Function<? super T, ? extends CompletableStage<U> > fn)`
- `thenApply` and `thenCompose` are analogous to `map` and `flatMap` in `Optional` and `Stream`
- `CompletableFuture` is a Functor as well as a Monad!

Callbacks via Chaining `CompletionStages`

- In summary...
- To create: use `runAsync` or `supplyAsync`
- `then<X><Y>` where
  - `X` is `Accept`, `Combine`, `Compose`, `Run`, ...
  - `Y` is nothing, `Both`, `BothAsync`, `Either`, `EitherAsync`, ...
- Have an awareness of different variants of the same method and use the appropriate ones
- In this module, we shall just stick with the simplest versions

Converting Synchronous to Asynchronous

- Give the following synchronous program fragment

```
int foo(int x) {
    if (x < 0) {
        return 0;
    } else {
        return new UnitTask(x).compute();
    }
}
```

- The asynchronous version is

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

Converting Synchronous to Asynchronous

- Suppose we have the following synchronous method calls

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

  which can be simplified to

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

- `bar` is defined as

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

- The equivalent asynchronous version is

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

Converting Synchronous to Asynchronous

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

```
int y = bar(x)
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

- To combine the results of two `CompletableFutures` via a `BiFunction`

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

- Both `fooAsync` and `barAsync` must be completed first, before the resulting `CompletableFuture` from invoking `thenCombine` can be completed


Multithreaded programming
Theory
Actual

Lecture Summary

- Appreciate asynchronous programming in the context of spawning threads to perform tasks in parallel
- Appreciate why busy waiting should be avoided
- Use of a callback to execute a block of code when an asynchronous task completes
- Encapsulating the context of asynchronous computations within `CompletableFuture`
- Able to convert synchronous code to an asynchronous version
- Refer to the Java API for a wide variety of chaining methods in the `CompletableFuture`/`CompletionStage` classes