CS2030 Lecture 12

Asynchronous Programming

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

Semester 1 2019 / 2020

Lecture Outline

- Synchronous programming
- Asynchronous programming
  - Thread creation
  - Busy waiting
  - Thread completion
- Callback
- `Runnable` versus `Callable` interfaces
- `Future` interface
- Promise using `CompletableFuture`

Synchronous vs Asynchronous Programming

- Given the following task unit

```java
import java.util.Random;

class UnitTask {
    int id;

    UnitTask () {
        this.id = new Random().nextInt(10);
    }

    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

```
class Sync {
    public static void main(String[] args) {
        System.out.println("Before calling compute()");
        new UnitTask().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
- The method might delay the execution of subsequent methods

Asynchronous Computation

- Create a thread that runs the `compute` method

```java
class Async {

    public static void main(String[] args) {

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

        Thread t = new Thread(() -> new UnitTask().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 (*cf.* `run()` method)

Busy Waiting

```
class Async {

    static void wait(int ms) {
        try {
            Thread.sleep(ms);
        } catch (InterruptedException e) { }
    }

    public static void main(String[] args) {

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

        Thread t = new Thread(() -> new UnitTask().compute());
        t.start();

        System.out.println("After calling compute()");

        while (t.isAlive()) {
            wait(1000);
            System.out.print(".");
        }

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

Busy Waiting

- Performing an unrelated task while waiting

```
public static void main(String[] args) {

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

    Thread t = new Thread(() -> new UnitTask().compute());
    t.start();

    System.out.println("After calling compute()");

    System.out.println("Do independent task...");
    wait(5000);
    System.out.println("Done independent task...");

    while (t.isAlive()) {
        wait(1000);
        System.out.print(".");
    }

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

Thread Completion via `join()`

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

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

    Thread t = new Thread(() -> new UnitTask().compute());
    t.start();

    System.out.println("Do independent task...");
    wait(5000);

    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`

Creating a Listener

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

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

Creating a Listener

```java
class Async implements Listener {

    void doAsync() {
        Thread t = new Thread(
                () -> {
                new UnitTask().compute();
                notifyListener();
            });
        t.start();
    }

    public void notifyListener() {
        System.out.println("compute() completed");
    }

    public static void main(String[] args) {
        Async async = new Async();
        async.doAsync();
        System.out.println("Do something else...");
    }
}
```

From `Runnable` to `Callable` Interface

- `Callable` runs in a separate thread and returns a value when completed, as well as allowing exceptions to be thrown
- SAM with abstract `<V> call()` method
- Pass `Callable` (or `Runnable`) to the `submit` method of an `ExecutorService`
  - `ForkJoinPool` is an implementor of `ExecutorService`
- `submit` method returns a `Future` object
  - Returns `Future<V>` (`Callable`) or `Future<?>` (`Runnable`)
- `Future`'s `get()` method waits for execution to complete and retreives result; returns `null` in the case of `Runnable`
- `get` method requires both `InterruptedException` and `ExecutionException` to be caught

Executing **Callable**

```java
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.Future;
import java.util.concurrent.ExecutionException;

class Async {
    public static void main(String[] args) {
        System.out.println("Before calling compute()");

        Future<Integer> future = new ForkJoinPool().submit(
                () -> new UnitTask().compute());

        try {
            System.out.println("Do independent task...");
            wait(5000);
            System.out.println("Done independent task...");

            Integer result = future.get();
            System.out.println("After compute(): " + result);
        } catch (InterruptedException | ExecutionException e) { }
    }
```

Useful Methods of `Future` Interface

- Other useful methods include:
  - `isDone()` returns `true` when the task completes
  - `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
  - `isCancelled()` returns true of the task has been cancelled
  - `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

From Future to Promise

- A future is a read-only container of a yet-to-exist result of an asynchronous computation
- A promise is a future that can be completed, hence in Java a promise is implemented as `CompletableFuture`
- From the perspective of the caller, the caller calls the asynchronous task and waits on the `Future` object until the result comes in — `Future` object is read-only
- Since `CompletableFuture` implements `Future`, the caller's perspective of a `CompletableFuture` does not change
- On the other hand, the callee can now construct a `CompletableFuture`, return it immediately to the caller and start the asynchronous task

Implementing Promise via `CompletableFuture`

- Using `CompletableFuture` as an implementation of `Future` with added completion logic

```
public Future<Integer> callee() {
    CompletableFuture<Integer> promise =
            new CompletableFuture<>();

    new ForkJoinPool().submit(() -> {
            int result = new UnitTask().compute();
            promise.complete(result);
    });
    return promise;
}
```

- The caller (e.g. `main`) is only aware of the `Future` object due to polymorphism

```
Future<Integer> future = new Async().callee();
```

Implementing Promise via `CompletableFuture`

- Unlike `Future`, a `CompletableFuture` may be explicitly completed by setting its value and status

```
public Future<Integer> callee() {
    CompletableFuture<Integer> promise =
            new CompletableFuture<>();

    new ForkJoinPool().submit(() -> {
            int result = new UnitTask().compute();
            promise.complete(result);
    });

    promise.complete(0);

    return promise;
}
```

- In the above, `promise.complete(0)` intervenes with an earlier completion

Encapsulating Asynchronous Logic

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

```java
public static void main(String[] args) {
    System.out.println("Before calling compute()");

    CompletableFuture<Integer> future = CompletableFuture
            .supplyAsync(() -> new UnitTask().compute());

    try {
        System.out.println("Do independent task...");
        wait(5000);
        System.out.println("Done independent task...");

        Integer result = future.get();
        System.out.println("After compute(): " + result);
    } catch (InterruptedException | ExecutionException e) { }
}
```

Callback via Chaining

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

```java
public static void main(String[] args) {
    System.out.println("Before calling compute()");

    CompletableFuture<Void> future =
        CompletableFuture
        .supplyAsync(() -> new UnitTask().compute())
        .thenAccept(s ->
                System.out.println("After compute(): " + s));

    System.out.println("Do independent task");
    wait(5000);
    System.out.println("Done independent task");
    future.join();
```

- Just like `get()`, the `join()` method is blocking and returns the result when complete

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
- Understand the difference between Java 5's `Future` and Java 8's `CompletableFuture`
- Encapsulating the context of asynchronous computations within `CompletableFuture`
- Take a first-hand look at the Java API for a wide variety of chaining methods in the `CompletableFuture` class; we will be discussing these soon