CS2030 Lecture 11

Asynchronous Programming

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

Semester 2 2019 / 2020

Lecture Outline

- Synchronous programming
- Asynchronous programming
  - Thread creation
  - Busy waiting
  - Thread completion
- Callback
- Encapsulating asynchronous logic with `CompletableFuture`

Timing the Execution of a Process

- To time the execution of a process,
  - `java.time.Instant`'s `now()` method returns the current `Instant` from the system clock
  - `java.time.Duration`'s `between()` returns the `Duration` of two `Instances` (an implementation of `Temporal`)
  - `Duration`'s `toMillis()`/`toNanos()`/... extracts the desired representation of the duration

```
java.util.Instant;
java.util.Duration;

Instant start, stop;
start = Instant.now();
/* perform some task */
stop = Instant.now();

long timeInMillis = Duration.between(start, stop).toMillis();
```

Execution Threads

- A computation task can be executed on a dedicated thread

```
Thread t = new Thread(...);
t.start();
```

- The `Thread` constructor takes in a `Runnable` which represents the computation by having it "run" within the abstraction `run()` method that takes in no arguments, and returns **`void`**.
- To retrieve the identity of the thread
  - `Thread.currentThread()`
  - `Thread.currentThread().getName()`
  - useful for thread debugging

Execution Threads

- `Thread.sleep(long millis)` causes the currently executing thread to sleep (i.e. temporarily cease execution) for the specified number of milliseconds
  - Used within a `try.. catch` block
  - Example, letting a thread sleep for one second

```
try {
    ...
    Thread.sleep(1000);
    ...
} catch (InterruptedException e) { }
```

- Useful for thread debugging in order to simulate threads having different computation loads

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

```java
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

```
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 functional interface with abstract method `run()`
- Start the thread with `start()` method

Busy Waiting

```java
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

```java
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

```
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

```
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...");
    }
}
```

CompletableFuture

- Use of `CompletableFuture` to encapsulate asynchronous logic
- static methods `runAsync` and `supplyAsync` creates `CompletableFuture` instances of `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>`

```
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
- 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

CompletableFuture is a both a functor and a Monad
to create: runAsync, supplyAsync
then<X><Y>:
X is Accept, Run, Combine, Compose
Y is nothing, Both, BothAsync, Either, EitherAsync