CSC301

Asynchronous patterns & Object pools
Announcements

● Check your A4 auto-marker results
  ○ In the auto-marker branch of your A4 repo (the original one, not your fork)
  ○ Please let me know if there is a mistake

● Team deliverable 2 is due Friday, March 10, at 4 pm
Term Test

- Next week, in class, 100 minutes long
- No questions about Git/GitHub, agile processes or product planning
- Only programming questions
  - Design patterns: Iterator, observer/observable, adapter, DAO, factory method, builder
  - Other topics: Object-oriented programming (with a focus on programming to interfaces), Serialization, lambda expressions, lazy-loading
  - Go over your individual assignments and code examples from lectures
- Questions will be similar in nature to the ones from past tests
  - Last term
  - Last year
Multi-Threading

● Multiple threads of execution running “simultaneously”
  ○ Well ... not exactly simultaneously, but sharing the CPU makes it seem like it
  ○ Java also supports creating new processes, but threads are much more common.

● Multi-threading is essential to most modern applications
  ○ Servers handling multiple simultaneous requests
  ○ GUI applications (GUI updates and background computations happen on separate threads)
  ○ Applications using 3rd party, remote API’s
    (applications cannot hang while waiting for network response)
Multi-Threading

- Multi-threaded programming is traditionally considered difficult
  - Easy to get things wrong
    - For example: Deadlock, race condition, starvation, etc.
  - Harder to reason about and debug
    - (mainly because the order of execution is nondeterministic)
Java & Multi-Threading

- The API’s related to threads (and concurrency) evolved over the years
- Initially, there was just `Thread`
  - Either extend it (i.e. create a subclass) or construct it with a `Runnable`
  - Once you create a Thread instance, you can `start` it
  - Fairly low-level methods allow synchronization and inspection of threads
Asynchronous Programming

- Say we have a (fairly long-running) task that returns a result
  - For example: Authenticate using a remote service (e.g. Google, Facebook, GitHub, etc.)
- As a programmer, I want to:
  - Run the task in a background thread, while doing other things in the main thread
  - Do something with the result of the task, whenever it’s ready
  - Handle any error that the task might throw
  - Specify a timeout, so that my program is not stuck waiting for the task, in case it runs indefinitely
Asynchronous Callbacks

- Instead of waiting for a function to return a result, tell it in advance what you want to do with its result.

- That is:
  - When calling the function, pass a *callback* (the callback is just another function)
  - The function runs in a separate thread
  - Once the function is done, the callback is called (with function’s result as its argument(s))
Synchronous vs. Asynchronous

Let's see the difference between synchronous and asynchronous code. 
(Note: This example uses JavaScript)

```javascript
try {
    data = syncFunc(someArg);
    // Do something with the data …
}
catch (error){
    console.log(error);
}

asyncFunc(someArg,
          function(data){
            // Do something with the data …
          },
          function(error){
            console.log(error);
          });
```
Asynchronous Programming

More features/requirements ...

- Chain tasks together
  - When a task is done, pass its result to the next task (and run the next task)
  - More complex “plumbing”, for example:
    - When a task is done, start three different tasks (in parallel)
    - Wait for multiple tasks to complete (all of them), and only then run the next task
    - Wait for the first of multiple tasks to complete, and pass its result to the next task
    - Pass exception down the chain, handle it and (possibly) continue to chain additional tasks

- Specify which threads to use for running background tasks
  - More control
Chaining Asynchronous Calls

In the example on the left, we chain two asynchronous functions.

Essentially, we are programming the following logic (plus error handling):

Call `asyncFunc1` and, when its result is ready, call `asyncFunc2`.
Chaining Asynchronous Calls

As our chains get longer (and, possibly, more complex), the code becomes very hard to manage.

For example:

```javascript
getData(function(a){
    getData(a, function(b){
        getData(b, function(c){
            getData(c, function(d){
                getData(d, function(e){
                    ...
                });
            });
        });
    });
});
```
Promises

- Asynchronous programming fits well with many of the tasks modern programmers deal with (because of the nature of the Internet)
- That being said, it doesn’t always fit well with the syntax that modern programmers are used to
  - Chaining multiple methods can result in code that is very hard to read
  - Exceptions need to be handled more carefully
- The *promise* design pattern to the rescue ...
Without promises

```javascript
asyncFunc1(someArg,
  function(result1){
    var x = doSomethingWith(result1);
    asyncFunc2(x,
      function(result2){
        doSomethingWith(result2);
      },
      function(error){
        console.log("f2 error: " + error);
      };
    },
    function(error){
      console.log("f1 error: " + error);
    };
  },
  function(error){
    console.log("f1 error: " + error);
  };
);```

With promises

```javascript
asyncFunc1(someArg
  .then(function(result1){
    return doSomethingWith(result1);
  })
  .then(function(result2){
    doSomethingWith(result2);
  })
  .catch(function(error){
    console.log("Caught " + error)
  });```

vs.
Promise

- *Promise* (aka *future*, *delay* or *deferred*) is an object that represents a result that is initially unknown
- Different languages use different constructs and terminology (e.g. Java uses *CompletableFuture*), but the basic idea is the same:
  - Decouple a computation from the value it produces
  - Compose computations (i.e. functions) together (before you run/execute them)
  - A nice [tutorial on using CompletableFuture](https://www.example.com/tutorial-completablefuture)
- Promises are based on a more general concept called *Monad*
  - Informally: A composable, generic type wrapper.
  - In our case: A Promise wraps an actual value, and can be composed with other promises
  - The formal definition is clearer in the context of functional programming
Java Threads - More Control

- Threads are resources (just like memory or CPU time)
- Sometimes we want more control over how resources are managed. For example:
  - Parallelism
  - Reusing threads (instead of creating new ones)
  - Managing cache (to optimize context switching)
  - etc.
Resource Pool

- **Object Pool** is a common design pattern (aka **Resource Pool**)
  - As the name suggests, an object that is responsible for managing a pool of resources
    - The resource pool manages the *lifecycle* of resources (e.g. construction, destruction, etc.)
  - Two of the most common examples for resources that can benefit from pooling:
    - Threads
    - Database connections
  - **ScheduledThreadPoolExecutor** is an example of a thread pool that comes built-in with Java
    - Here is [really nice tutorial](https://example.com) you might find useful
Code Example

- This commit history of this short code example shows a few of the concepts discussed in this lecture.
Summary

- We use multi-threading to perform multiple tasks simultaneously.
- We can use *thread-pools* to manage our threads.
  Common example:
  - The UI thread (a pool with a single thread)
  - Worker threads (use for long-running background tasks)
- When working with asynchronous code:
  - Use *promises* to chain functions together (i.e. create pipelines) and keep the code easily maintainable.
  - When executing an asynchronous pipeline, use a thread-pool for long-running, background tasks (e.g. network request, or reading from disk).