Lab 1: Playing with the Timeclock command line

Please note that you also have a Timeclock Cheat Sheet in your course packet.

Try some free-form fun with the timeclock interface. Try out different commands. Spend some time making mistakes. What happens if you try to stop a job that wasn't started? Start a job that doesn't exist?

As you type along, you'll likely make some different kinds of mistakes:

- You might forget quotes where they're needed.
- You might leave out a comma that's required.
- You might mistype a command name.

Some of those mistakes will produce much less pleasant error messages than the ones from the mistakes in the first paragraph. You'll learn more about the "why" behind those messages later. For now, just don’t be alarmed when you see them. In fact, spend a little time going out of your way to cause them. Save up the error messages and make sure that you understand them by the end of the day (getting us to explain them, if necessary).

Lab 2: Testing command results

As you just saw, your tests will use the utility function `assert_equal.` Where is it on your computer? We're going to be mean and make you type it in. (That's because we believe most people learn a language fastest when they start typing in it, not when they watch other people type. So typing this definition will help you understand it.

Put the following in a file called `asserts.rb`.

```
def assert_equal(expected, actual)
  if expected != actual
    puts "FAILURE: Expected <#{expected}>, but was <#{actual}>."
  end
end
```

At this point, you could do one of two things. You could believe you typed everything correctly and proceed to testing Timeclock. Or you could check. We recommend checking.

One of the nice things about languages, like Ruby, that have interpreters is that they make it easy to check things like that. Use `start.bat` to start up the Ruby interpreter. Do it in the same directory you created `asserts.rb`. Then type this:

```
load 'asserts.rb'[1]
assert_equal('hi', 'hi')
assert_equal('hi', 'there')
```

Does the first one succeed (print nothing)? Does the second one fail? If something odd happens, try to figure out where you made a mistake, and don't hesitate to ask for help.

Now that you trust `asserts.rb`, create your test file. Call it `test.rb`. It should start out looking like this:

```
require 'loadable-timeclock'
require 'asserts'
```

Now you can add your tests after those lines, like this:

```
assert_equal("Job 'timeclock' created.", job('timeclock'))
```

Run the tests like this:

```
C:\tutorial> ruby test.rb
```

Try some more tests of the `job` command. After a while, we suggest you try the `start` command. You should quickly run into a problem testing. That'll be a good time for us to stop and discuss something new, *regular expressions*. Then we'll resume the lab.

[1] What's `load`? What happened to `require`? The main difference is that `load` always loads the file named, but `require` only loads it once. Later, you're going to change `asserts.rb`. You might want to try those changes in the interpreter, which you might have left running. To load the new version, you have to use `load`, so you might as well use it here.

After the lab resumes, you'll want to add this to `asserts.rb`:

```
def assert_match(expected_regex, actual_string)
  unless expected_regex =~ actual_string
    puts "FAILURE: Expected <#{actual_string}> to match <#{expected_regex.inspect}>."
  end
end
```

Lab 3: Test::Unit

Your job here is just to explore the Test::Unit framework. You'll be using it more later.

Start by changing the beginning of your `test.rb` file to require `test/unit` instead of `asserts.rb`:

```
require 'test/unit'
```

Next, "wrap" the following around your test statements:

```
class TimeclockTests < Test::Unit::TestCase

  def test_job
    all your test statements go here
  end
end
```

Run that (`ruby test.rb`). See if it works.

Now play around with your tests. Try tests that fail. What happens to `assert_equal` statements after the failure? Are they tried? (The output says how many assertions were tried. Is it different in the case of failure?)

Try splitting your tests into several different functions (methods). In what order are tests run?

What if one of the function's names doesn't start with "test"? Of what use could such a function be?

Lab 4: Two new classes

Begin by walking through the examples for Arrays and Hashes in the Ruby Cheat Sheet (approximately pages 4 and 5).

"Walk through?" That means use the Ruby interpreter to type those expressions. As we've seen, the Timeclock program is a ruby interpreter. But it's a slightly modified one, one that prints results in a way that won't be useful for this lab. (It doesn't show the quotes around strings or the braces around arrays and hashes.)

So start the interpreter by typing `irb` at your command prompt. Now go to page 4 of the Ruby Cheat Sheet.

Now try a few other things.

1. Put arrays inside arrays. For example, how would you convert `[1, 2]` to `[ ['hi'], 2, ['you', 'and me'] ]` ?
2. If the above array is the value of variable `array`, what's the value of `array[0]`? What's the value of `array[0][0]`?

   What's the value of `array.last`? Of `array.last.last`?
3. Make a dictionary that associates `'bret'` with `'texas'` and `'brian'` with `'illinois'`. Have the variable `states` refer to it.

   Make an empty array.

   How can you put the dictionary into the array?

   Once you've got it there, what's an expression that shows Bret's home state?

   Have the variable `instructors` refer to this array: `['bret', 'brian'].` (Can you get that array from the dictionary `states`?) Write an expression that tells you in what state the second instructor lives.

Lab 5: Web Services

The Timeclock Web Services Interface *defines the different messages that the server accepts. You've seen several of them demonstrated already.*

Improving the sample tests

You just finished seeing a demonstration of creating two tests. You can find something close to those tests in `web-test.rb`.

Consider the first test. It's kind of inadequate, since `start` does more than just create a record:

- It's supposed to start the job running. You can check that with `session.running?('star')`.
- The record `start` creates is supposed to contain the job it's for. That job's name had better be the same string as the argument to `start`. You can get the record's job with `record.job` and the job's name with `job.name`.
- `start` is supposed to return the job it started. That job had better be named `'star'`.

Update the test to check whether all these things that are supposed to be true are.

Then update the second test to check that an incorrect starting-of-a-started-job leaves alone things it should leave alone (like whether 'star' is running).

Run wild!

The *Timeclock Web Services Interface* document defines a number of timeclock commands. They should seem familiar because most of them straightforwardly correspond to command-line commands. Write tests for them.

It's probably easiest to write tests for various combinations of starting, pausing, and stopping jobs. Check records and which jobs are running. Write utility functions as you find them useful.

Try lots of different things. Ask questions when you want to do something but don't know how.