In most of the United States (which does observe DST), the answer is no. In March, we'll have a day with 23 hours (spring forward), and in November we'll have a day with 25 (fall back). This means arithmetic won't always work as you expect; 1:45 a.m. plus 30 minutes might equal 1:15, for instance.

But we've tested any time-sensitive code on those boundary days, right? For locations that honor DST and for those that do not?

Oh, and don't assume that any underlying library handles these issues correctly on your behalf. Unfortunately, when it comes to time, there's a lot of broken code out there. And leap seconds *do* make a difference.

Finally, one of the most insidious problems brought about by time occurs in the context of concurrency and synchronized access issues. It would take an entire book to cover designing, implementing, and debugging multithreaded, concurrent programs, so we won't take the time now to go into details, except to point out that most code you write in most languages today *will* be run in a multithreaded, multiprocessor environment (see the section on page 200 for an interesting "gotcha" in C#).

So ask yourself, what will happen if multiple threads use this same object at the same time? Are there global or instance-level data or methods that need to be synchronized? How about external access to files or hardware? Be sure to add the `lock` keyword to any property or method that needs it, and try firing off multiple threads as part of your test.

5.8 Try It Yourself

Now that we've covered the Right BICEP and CORRECT ways to come up with tests, it's your turn to try.

For each of the following examples and scenarios, write down as many possible unit tests as you can think of.

Exercises

1. **A simple stack class.** Push `String` objects onto the stack, and `Pop` them off according to normal stack semantics. This class provides the following methods:

Answer on 212

```
using System;
public interface StackExercise {
  /// <summary>
  /// Return and remove the most recent item from
  /// the top of the  stack.
  /// </summary>
  /// <exception cref="StackEmptyException">
  /// Throws exception if the stack is empty.
  /// </exception>
  String Pop();

  /// <summary>
  /// Add an item to the top of the stack.
  /// </summary>
  /// <param name="item">A String to push
  /// on the stack</param>
  void Push(String item);

  /// <summary>
  /// Return but do not remove the most recent
  /// item from the top of the stack.
  /// </summary>
  /// <exception cref="StackEmptyException">
  /// Throws exception if the stack is empty.
  /// </exception>
  String Top();

  /// <summary>
  /// Returns true if the stack is empty.
  /// </summary>
  bool IsEmpty();
}
```

StackExercise.cs

Here are some hints to get you started: What is likely to break? How should the stack behave when it is first initialized? How should it behave after it has been used for a while? Does it really do what it claims to do?

2. **A shopping cart.** This class lets you add, delete, and count the items in a shopping cart.

Answer on 213

What sort of boundary conditions might come up? Are there any implicit restrictions on what you can delete? Are there any interesting issues if the cart is empty?