# CITS5501 Software Testing and Quality Assurance
# Semester 1, 2020
# Workshop week 8 – Data-driven testing

## Testing properties of methods

Sometimes, in addition to testing a function or method with known test cases, we would like to test it with *random* values as well. This can give us more confidence that our method is behaving sensibly.

For instance, consider the method

```
public static double sin(double x)
```

which calculates the sin of a number. We will almost certainly have some specific test cases developed for testing this method.

But in addition, we might test it on many randomly generated `double`s. We don't know in advance what their sin will be. But we *do* know that, whatever they are, they should satisfy the following condition:

- The output of sin is always between -1.0 and 1.0.

So we could gain more confidence that our method behaves properly on a wide range of inputs using something like the following test code:

```
// ...
import java.util.Random;
public class TrigTest {
  @Test
  public void sinOutputIsWithinSensibleRange() {
    Random r = new Random();
    for (int i = 0; i < 10000; i++) {
      double d = r.nextDouble();
      double result = sin(d);
      assertTrue("result " + result + " should be >= -1", result >= -1);
      assertTrue("result " + result + " should be <=  1", result <=  1);
    }
  }
}
```

This is an example of *data-driven* testing – we apply the same basic test, to many bits of input data – as well as *randomised testing* (testing on randomly-generated input values). In this case, we have identified an *invariant property* of the sin method – it always should

produce values in the range (-1,1) – and randomized, data-driven testing helps give us confidence that this invariant holds.

Other examples of invariants:

- When we apply the method `ArrayList sort(ArrayList someList)` to a list, the returned list should always have the same length as the input list.
- When we apply the method `ArrayList sort(ArrayList someList)` to a list, the returned list should always have the same members as the input list (but the order may be different).
- When we apply the method `ArrayList sort(ArrayList someList)` to a list, each element of the list should be greater than or equal to the element before it.

You might notice that these invariants also all happen to be *postconditions* of the method. Postconditions are a good source of testing invariants. And conversely, coming up with invariants can help us improve the documentation for our method, by helping us more precisely specify the postconditions.

But sometimes our invariants are more complicated than just being preconditions. For instance:

- If we apply `reverse(reverse(someString))` to a string, we should end up with the original string.

### Exercise

See if you can come up with some invariants for the following methods, which could be tested using randomised testing.

1. The method `String stripSpaces(String aString)`, which removes all spaces from a string.
2. The method `String shuffle(String aString)`, which takes as input a string, and returns a new string containing the same letters, but in a random order.
3. The method `PngImage rotateClockWise(PngImage anImage)`, which takes as input an object of type `PngImage` (representing a PNG image), and outputs the image rotated 90 degrees clockwise.
4. The method `String[] splitOnSpaces(String aString)`, which takes a string as input, splits the string wherever it sees a space character, and returns an array of the resulting strings.

   For example, splitting the string `"nice weather"` produces the array `{"nice", "weather"}`.
5. The method `void replace(WordDocument doc, String searchString, String replaceString)`. This takes as input a `WordDocument` object; it searches in the document for all instance of the String `searchString`, and replaces them with `replaceString`. There is no return value; the document is mutated "in place".

If you have time, consider how you would write JUnit-based tests for these. Some can be written quite simply using Java's built-in libraries and a `for` loop (as shown in the

example); others might be more difficult.

A testing library dedicated to *property-based testing* provides a number of features that help with this style of testing:

- the ability to easily generate random data structures (for example, Word documents).
- the ability to concisely express properties that we want to hold.
- the ability to *shrink* failing examples. For instance, suppose we randomly generate a very large Word document, test the `replace()` method shown above, and discover that it fails. It may be difficult to work out where in our code the problem is. *Shrinking* means that the testing library tries smaller versions of the failing Word document, and tries to give us the *smallest example that fails*. This can help greatly in identifying the problem.

### Self-study

Take a look at `junit-quickcheck`, a property-based testing library for Java.

The documentation is here:

- https://pholser.github.io/junit-quickcheck/site/0.9.1/

And the GitHub repository for it is here:

- https://github.com/pholser/junit-quickcheck

See if you can formulate any of the tests from the exercise using `junit-quickcheck`.