One of the tools that we can use to experimentally assess the efficiency of programs is quite straightforward – just measure the actual time taken to run the program. This is known as *wall-clock time*. Perhaps however this is not as simple as it seems. In this lab you'll work with a simple timer setup to see how this method works in practice, and to become aware of some of the problems that can arise.

A simple timer class `Timer.java` is provided. This functions something like a stop-watch - you start and stop an instance of a timer and it reports the time taken (for convenience it also maintains a list of all the times it has recorded). The times recorded by this class are measured in nanoseconds, i.e., one-billionths of a second and are returned as `long` values. To obtain output that's comprehensible to humans, you could scale the nanosecond values by a suitable factor, for instance dividing by one million to produce a result in milliseconds. You should implement a `getTimeInMs` method to perform this function, and also a `getTimeInS` to compute the result in seconds (since that's the SI unit for time). Think carefully about what return type would be appropriate and how the calculation should be done. Have a read of the rest of the `Timer` code while you're there.

Experiment 1

The skeleton outline for this experiment includes a method `sumRnd(int n)` is provided that returns the sum of n random double values using a simple for loop. You can use a `Timer` object to record the time taken when this method is called.

1. What is the (typical) ratio between the time taken for `sumRnd(1000)` and `sumRnd(100)`. Does that make sense? What *should* the ratio be?
2. How large do you need to make n before the ratio between the time taken for `sumRnd(10*n)` and `sumRnd(n)` approaches what it *should* be?
3. When you reach that point, how much total time (roughly) is being used?
4. What does this say about the design of wall-clock experiments to test efficiency?

Experiment 2

This experiment illustrates some odd behaviour of string construction in Java. A common situation that arises in many programs is that a long string (e.g., the content of a log file) is gradually put together from individual pieces.

The skeleton outline for this experiment includes three methods for constructing a random string of n lower case letters.

1. Read the code for those methods - what are the differences? Which one would be the most likely for you to write if we'd asked you to?
2. Compare the time required by the three methods for various values of n. Are they always similar? Remember to make n large enough that significant time is required.
3. The first method (using basic string concatenation) slows down *a lot* as n gets large. Why should that be? What does it say about using string concatenation in programs in general?

Reflection and extension

- Review the javadoc for System.nanoTime(), and java.util.Random. Both are useful in experimentation and testing.
- In Assignment 1 you're going to be asked to report on the timing outcomes of various experiments. To get sensible results you'll need to perform the experiments multiple times and then analyse the output (most likely in a spreadsheet). This suggests a number of things to think about:
  - How can you structure your experimental code so that it runs multiple times without needing to restart it from the terminal each time?
  - How could you arrange that the output is suitable for immediate copy-pasting from the terminal into a spreadsheet?
  - More ambitiously, could you set up an experimental harness that writes output to a file format that can be imported directly?