

Assignment	<ol style="list-style-type: none"> 1. Write a Stopwatch class that can be used to compute elapsed time. Use the package stopwatch for the source code. 2. Rewrite the SpeedTest to eliminate duplicate code and be easier to reuse. 3. Explain the results of running the 5 tasks.
What to Submit	<ol style="list-style-type: none"> 1. Create a repository named stopwatch (lowercase) on Github/Bitbucket. 2. Commit your source code. Please do not commit the "bin" directory. Use a <code>.gitignore</code> file to prevent this. 3. Write a good <code>README.md</code> that describes project and answers question 2-3.
Evaluation	<ol style="list-style-type: none"> 1. Correctness of code. 2. Quality of code, including Javadoc and code format. 3. Quality of your explanation for problem 3. 4. Quality of your solution to problem 4.
Individual Work	Do this assignment individually. You may discuss <i>ideas</i> for solution, but not share code. You can ask TA's questions or for specific problems. All submitted work must be your own.

1. Write a Stopwatch

Write a Stopwatch class that computes elapsed time between a start and stop time. Stopwatch has 4 methods:

`start()` reset the stopwatch and start if if stopwatch is not running. If the stopwatch is *already* running then `start` does nothing.

`stop()` stop the stopwatch. If the stopwatch is *already* stopped, then `stop` does nothing.

`getElapsed()` return the elapsed time **in seconds** with decimal. There are 2 cases:

- (a) If the stopwatch is running, then return the elapsed time since `start` until the current time.
- (b) If stopwatch is stopped, then return the time between the start and stop times.

`isRunning()` returns true if the stopwatch is running, false if stopwatch is stopped.

How to Compute the Time: Java has two methods to get the current time :

`System.nanoTime()` returns the current time in nanoseconds (long). One nanosecond is 1.0E-9 second. This is the most accurate method. The time may "wrap" back to 0 (if you are *really* unlucky).

`System.currentTimeMillis()` returns the current time in milliseconds (=1.0E-3 sec).

Your Stopwatch should use **`System.nanoTime()`** since it is more accurate.

Example:

```
Stopwatch timer = new Stopwatch( );
System.out.println("Starting task");
timer.start( );
doSomething( );           // do some work
```

Stopwatch

```
+ getElapsed( ) : double
+ isRunning( ) : boolean
+ start( ) : void
+ stop( ) : void
```

```
timer.stop( );           // stop timing the work
System.out.printf("elapsed = %.6f sec\n", timer.getElapsed() );
if ( timer.isRunning() ) System.out.println("timer is still running!");
else System.out.println("timer is stopped");
```

Output:

```
Starting task
(doing work)
elapsed = 0.021188 sec
timer is stopped
```

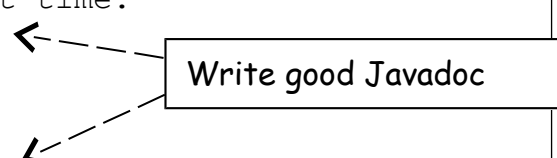
Template for Stopwatch

```
package stopwatch;
/**
 * A Stopwatch that measures elapsed time between a starting time
 * and stopping time, or until the present time.
 * @author Bill Gates
 * @version 1.0
 */
public class Stopwatch {
    /** constant for converting nanoseconds to seconds. */
    private static final double NANOSECONDS = 1.0E-9;
    /** time that the stopwatch was started, in nanoseconds. */
    private long startTime;
    //TODO you need another attribute or two.

    //TODO Implement constructor and methods

    /** Start the stopwatch if it is not already running. */
    public void start( ) { }

    . . . // more methods
}
```



2. Run the SpeedTest tasks and explain the results

The file **SpeedTest.java** (in class **week2** folder) contains tasks to compute speed of common Java operations like appending to a String and adding numbers. In the main method it times 6 tasks.

1. append 50,000 characters to a String. Display *length* of the result (Don't display the string, of course)
2. append 100,000 characters to a String. Display length of the result. Since this is twice as many chars as task 1, does it take 2X as much time? Explain why this task takes so long.
3. append 100,000 characters to a StringBuilder, then convert result to String and display its length.
4. sum 1,000,000,000 **double** values from an array.
5. sum 1,000,000,000 **Double** objects having the same values as in task 4.
6. sum 1,000,000,000 **BigDecimal** objects having the same values as in task 4.

Run the class and record the times in README.md.

Then explain *why* some tasks take longer.

- Why does appending 100,000 chars to a String take more than 2X the time to append 50,000 chars?

- Why is appending to `StringBuilder` so much different than appending to `String`? What is happening to the `String`?
- Explain difference in time to sum `double`, `Double`, and `BigDecimal`. Which is faster and why?

Note:

For tasks that sum array values, `SpeedTest` creates an array of values *before* it starts the Stopwatch, so that the time to create objects is not included in the time measured when we run the task. The array size is less than the loop count to avoid running out of JVM memory.

3. Create a README.md file to explain the results

Create a **README.md** file in your repository. In this file, write the **times reported** for running each task and explain the differences. You should explain:

- why is there such a big difference in the time used to append chars to a `String` and to a `StringBuilder`? Even though we eventually "copy" the `StringBuilder` into a `String` so the final result is the same.
- why is there a significant difference in times to sum `double`, `Double`, and `BigDecimal` values?

Example README.md

This file can contain formatting using Markdown syntax.

```
# Stopwatch tasks by Bill Gates (60105400000) (*)

I ran the tasks on a Microsoft Surface Pro (of course), and got
these results:

Task                                     | Time
-----|-----:
Append 50,000 chars to String           | 3.8888 sec
Append 100,000 chars to String           | 8.1230 sec
Append 100,000 chars to StringBuilder    | 2.0880 sec
Add 1 billion double using array         | 1.3205 sec
etc.                                     | ...

## Explanation of Results

I have no idea why the times are different.
```

(*) That number is his net worth in US\$, not his student id.

Markdown is a simple text formatting syntax that is used on Github, Bitbucket, and many other places. You should know how to use it. For an intro to Markdown syntax see:

- <https://bitbucket.org/tutorials/markdowndemo>
- <http://dillinger.io/> (online markdown editor to practice)

4. Rewrite the Stopwatch Tasks **This is the most important part!**

The sample code for these tasks contains a lot of **duplicate code**. Rewrite the **SpeedTest** to eliminate the duplicate code and create a reusable **TaskTimer** class! Here are the guidelines:

- "Let's Eliminate Duplicate Code" by Thai: <http://goo.gl/TGiUqC> Explains why and how to solve this problem.
- Apply the principle: "Separate the part that varies from the part that stays the same." Then, "encapsulate the part that varies." In each of the 5 tasks in **SpeedTest**, what part of each task is the same (duplicate code)? What part of each task is different? (Separate this part into its own class and make it *Runnable*.)
- Create a **TaskTimer** class that will compute and print the elapsed time for any task, without any duplicate code. This is described in Thai's write-up.
- Create a separate Java class for each of the 5 tasks, using the design from Thai's write-up. Each task class should:
 - implement **Runnable**
 - the **run** method performs the task, but does not print the task description! Let the task runner do that.
 - have a **toString()** method that describes the task.
 - have a constructor to initialize parameters needed by the task. Also initialize any data that we don't want the Stopwatch to time!

Write a Main class that creates task objects, creates a **TaskTimer**, and then uses **TaskTimer** to run the tasks.

When you finish, your code should have (almost) no duplicate code and use polymorphism.

