Get + Set

The getters and setters in the moment library are more complex then standard getters and setters. They have some unique properties and behaviours that we will describe before delving into the testing methodology and results.

* The same method is used for both getting and setting. If no input is given it is used as a getter, otherwise it is used as a setter.
* Every method with the exception of year() has the ability to “bubble up” to the next time unit if the standard range is exceeded (ex. greater than 59 seconds will increase minutes and less than 0 seconds will decrease minutes)
* Decimal values are valid inputs to the function, but they do not comprise their own equivalence classes because the methods simply truncate them and treats them as integers.
* Numerical Strings can be used as valid input; the setters will simply use its numerical value.
* There exist generic getters and setters that accept a type parameter in addition to the standard value parameter.

Since the use of getter and setter methods is simple functionality combined with only one input, we decided to establish equivalence classes to assist in the proper testing of the methods. This equivalence class could be reused for the different units of time with few modifications since for the most part they shared the same pool of valid and invalid inputs.

|  |  |
| --- | --- |
| **Valid Equivalence Classes** | **Invalid Equivalence Classes** |
| Empty Input | Non Integer or Float Input |
| Between Upper Bound and Lower Bound | Non-Numerical String |
| Less than Lower Bound |  |
| Greater than Upper Bound |  |
| Numerical String Input (ex. “11”) |  |

There are two exceptions to this table for specific time units.

* Since years have no unit to bubble up to, they are capped between -270000 and 270000 and will not accept inputs outside of that range.
* Day of Week and Month setters will additionally accept partial and full strings like “Mon”, “Monday”, “Mar” and “March” respectively.

Test cases were constructed using both the above table and the exceptions described above. As all of the time units share a similar set of equivalence classes, we will only discuss test cases for one unit of time (seconds in this case) as an example. The date used in the tests is initialized in the form YYYY-MM-DD HH:mm:ss:SSS as 2013-02-08 09:30:26.123.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test ID** | **Description** | **Input** | **Expected** | **Actual** |
| 1 | Calls seconds as a getter. | N/A | 26 seconds | 26 seconds |
| 2 | Calls seconds as a setter with input within the bounds | 43 | 43 seconds | 43 seconds |
| 3 | Calls seconds as a setter with input above the bounds | 88 | 28 seconds 31 minutes | 28 seconds 31 minutes |
| 4 | Calls seconds as a setter with input below the bounds | -13 | 47 seconds  29 minutes | 47 seconds  29 minutes |
| 5 | Calls seconds as a setter with a floating point number | 52.7 | 52 seconds | 52 seconds |
| 6 | Calls seconds as a setter with numerical string | “54” | 54 seconds | 54 seconds |
| 7 | Calls seconds with an invalid input | “BadInput” | 26 seconds | 26 seconds |

As the table clearly shows, the tests for most of the time units passed completely. There were however two outstanding issues of inconsistency between time units found through testing.

* Three time units: Day of Week, Day of Year and Quarter handle floating point inputs differently than the standard time units. The standard method of handling floating points is to truncate the decimal and treat the input as an integer. The three time units mentioned above will instead round the decimal appropriately.
* Two time units: Day of Week and Quarter react oddly to invalid inputs. While most time units simply leave the date unmodified when given invalid input, Day of Week while change the date to am arbitrary value and Quarter will corrupt the date entirely, causing it to become NaN.

Analysis of the source code revealed the cause of these inconsistencies. While the standard units of time all use the same generic getter and setter function behind the scenes, the time units mentioned above use their own built-in getters and setters. This is understandable since they are all derived time units that are not directly stored in the date as the base ones like hours and minutes are. What is not clear is why the implementation of them does not follow the same convention as the base time units in the case of floating point and invalid inputs. This issue could cause difficulties with user who expect the library to behave the same way for all time units, since these inconsistencies are not mentioned in the documentation.