# WSNSimulation API Usage

**Zack Rauen**

**Classes** in the package:

1. **Complex** – Represents complex numbers (Note: this is a simplified version for use in WSNSimulation)
    I. Constructors
        i. Complex()
            ⇒ Sets real and imaginary parts to 0
        ii. Complex(double real, double im)
            ⇒ Sets the real to real and the imaginary to im
    II. Get Functions
        i. double : getReal
            ⇒ Returns the real portion of the Complex number
        ii. double : getImaginary
            ⇒ Returns the imaginary portion of the Complex number
        iii. double : getMagnitude
            ⇒ Returns the magnitude of the Complex number
        iv. double : getPhase
            ⇒ Returns the phase of the Complex number
    III. Set Functions
        i. setReal(double)
            ⇒ Sets the real portion of the Complex number
        ii. setImaginary(double)
            ⇒ Sets the imaginary portion of the Complex number
        iii. setValues(double real, double im)
            ⇒ Sets the real to real and imaginary to im
    IV. Others
        i. String : toString
            ⇒ Returns the String representation of the Complex number. Note: this does not include any formatting/rounding.
        ii. String : toFormattedString
            ⇒ Returns the String representation of the Complex number rounded to two digits past the decimal for each portion of real and imaginary.
2. **Sensor** – Represents sensors in the simulation. Each Sensor has an ID, which is auto incremented in this case, a Complex alpha value, a Complex H value and a Complex N value.
    I. Constructors
        i. Sensor(int id, Complex alpha, Complex h, Complex n)
            ⇒ Sets the values for the sensor
    II. Get Functions

i. Int : getID
ii. Complex : getAlpha
iii. Complex : getHVal
iv. Complex : GetNVal

III. Set Functions
i. setID(int)
ii. setAlpha(Complex)
iii. setHVal(Complex)
iv. setNVal(Complex)

IV. Others

3. **SetupException** – Exception for the SimulationSetup inherits RuntimeException, only thrown when an error that would break the simulation occurs, so it is an unchecked exception (meaning you aren't required to use try catch block).
I. Constructors
i. SetupException(SetupException.ExceptionType)
⇒ Allows for an easier way of checking exactly why there was an error.

II. Get Functions
i. SetupException.ExceptionType : getExceptionType
⇒ Return the exception type so the user can correct the error

III. Set Functions

IV. Others

4. **Simulation** – Represents a single simulation
I. Constructors
i. Simulation(SimulationSetup)
⇒ This constructor will automatically run the simulation based on the setup provided.

II. Get Functions
i. Complex : getVVal
ii. Complex : getYVal
iii. Complex : getThetaHat
iv. List<Sensor> : getSensorList
⇒ Returns the list of sensors that is automatically generated upon calling the constructor
v. Sensor : getSensor(int)
⇒ Shorthand for getSensorList().get(int)
vi. SimulationSetup : getSetup

III. Set Functions

IV. Others
i. sortSensorsByNumber(Boolean descending)
ii. sortSensorsByAlpha(Boolean descending)
iii. sortSensorsByH(Boolean descending)

iv. sortSensorsByN(Boolean descending)

5. **SimulationSetup** – Represents the setup for the simulation
   I. Constructors
      i. SimulationSetup()
         ⇒ Sets up the default settings
      ii. SimulationSetup(args[])
         ⇒ Sets up the settings by name
   II. Get Functions
      i. double : getC
      ii. String : getObservation
      iii. double : getTheta
      iv. int : getSensorCount
      v. double : getPower
      vi. double : getVarianceN
      vii. double : getVarianceV
      viii. double : getK
      ix. Boolean : isUniform
      x. Boolean : isOptimum
         ⇒ Shorthand for !isUniform()
      xi. Boolean : isRician
      xii. Boolean : isAWGN
         ⇒ Shorthand for !isRician()
      xiii. double : getSensingSNR
      xiv. double : getChannelSNR
   III. Set Functions
      i. setObservation(String)
      ii. setTheta(double)
      iii. setSensorCount(int)
         ⇒ Throws SensorException for negative numbers
      iv. setPower(double)
         ⇒ Throws SensorException for negative numbers
      v. setVarianceN(double)
         ⇒ Throws SensorException for negative numbers
      vi. setVarianceV(double)
         ⇒ Throws SensorException for negative numbers
      vii. setK(double)
         ⇒ Throws SensorException for negative numbers
      viii. setUniform(Boolean)
         ⇒ Throws SensorException for if channels are set as Rician
      ix. setOptimum(Boolean)
         ⇒ Shorthand for setUniform(!input)

x. setRician(Boolean)
⇒ Will automatically switch to Optimum alphas if Rician is chosen
xi. setAWGN(Boolean)
⇒ Shorthand for setRician(!input)

IV. Others
i. addListener(SetupListener)
ii. removeListener(SetupListener)

6. **SimulationManager** – Helps to manage all things in the simulation and setup
I. Constructors
II. Get Functions
i. Simulation : getLastSimulation
ii. SimulationSetup : getSimulationSetup
iii. SetupListener : getListener
III. Set Functions
i. setSetupListener(SetupListener)
IV. Others
i. getInstance
⇒ SimulationManager is a singleton
ii. runSimulation
⇒ This runs the Simulation and sorts the sensors based on the most recent way of sorting them
iii. updateSimulation(SharedPrefs)
⇒ This is only used in android, this will save the current setup to the shared preferences so it can be loaded in later
iv. sortSensors
⇒ Sorts the sensors the same way as previously done in case of shuffling etc.
v. sortSensors(int)
⇒ Sort by a specific field. 0 for ID, 1 for Alpha, 2 for H and 3 for N. This is very useful for swapping between ascending and descending because SimulationManager remembers what was done. For instance calling sortSensors(0) will sort the sensors by ID ascending. Calling sortSensors(0) again will sort the sensors by ID descending. This cycle repeats or starts over when a new field is used.
vi. sortSensorsByNumber(Boolean descending)
vii. sortSensorsByAlpha(Boolean descending)
viii. sortSensorsByH(Boolean descending)
ix. sortSensorsByN (Boolean descending)

**Interfaces** in the package:

1. **SetupListener** – Alerts the listener for a change in the simulation

**Use Cases:**

1. Adjust the setup
    1.1. Use SimulationManager to retrieve the current setup and then use the set* functions to adjust the setup.

```
SimulationManager.getSimulationSetup().setObservation(SimulationSetup.DEFAULT_OBSERVATION);
SimulationManager.getSimulationSetup().setSensorCount(SimulationSetup.DEFAULT_SENSOR_COUNT);
SimulationManager.getSimulationSetup().setTheta(SimulationSetup.DEFAULT_THETA);
SimulationManager.getSimulationSetup().setPower(SimulationSetup.DEFAULT_POWER);
SimulationManager.getSimulationSetup().setVarianceN(SimulationSetup.DEFAULT_N);
SimulationManager.getSimulationSetup().setVarianceV(SimulationSetup.DEFAULT_V);
SimulationManager.getSimulationSetup().setK(SimulationSetup.DEFAULT_K);
SimulationManager.getSimulationSetup().setRician(SimulationSetup.DEFAULT_RICIAN);
SimulationManager.getSimulationSetup().setUniform(SimulationSetup.DEFAULT_UNIFORM);
```

2. Get values from the setup
    2.1. Use the SimulationManager to retrieve the current setup and then use the get* functions to get any desired value.

```
DecimalFormat two = new DecimalFormat("#.##");
System.out.println("Current Setup:"); System.out.println("");
System.out.println("Observation: "+SimulationManager.getSimulationSetup().getObservation());
System.out.println("Theta: "+two.format(SimulationManager.getSimulationSetup().getTheta()));
System.out.println("Number of sensors to use: "+SimulationManager.getSimulationSetup().getSensorCount());
System.out.println("Power Value: "+two.format(SimulationManager.getSimulationSetup().getPower()));
System.out.println("N-Variance: "+two.format(SimulationManager.getSimulationSetup().getVarianceN()));
System.out.println("V-Variance: "+two.format(SimulationManager.getSimulationSetup().getVarianceV()));
System.out.println("Alpha Type: "+(SimulationManager.getSimulationSetup().isUniform() ? "Uniform" :
"Optimum"));
System.out.println("Channel Type: "+(SimulationManager.getSimulationSetup().isRician() ? "Rician" :
"AWGN"));
System.out.println("K-Value: "+(SimulationManager.getSimulationSetup().isRician() ?
two.format(SimulationManager.getSimulationSetup().getK()) : "N/A"));
```

3. Run the simulation
    3.1. Simply call *SimulationManager.runSimulation()*
4. Run the simulation multiple times
    4.1. Call *SimulationManager.runSimulation()* multiple times

```
for (int i = 0; i < runCount; i++) {
    SimulationManager.runSimulation();
}
```

5. Get values from the simulation
    5.1. Use SimulationManager to obtain the most recent Simulation and use the get* functions to get desired values

```
System.out.println("Current Simulation:"); System.out.println("");
System.out.println("Theta Hat: "+SimulationManager.getLastSimulation().getThetaHat().toFormattedString());
System.out.println("Y Value: "+SimulationManager.getLastSimulation().getYVal().toFormattedString());
System.out.println("V Value: "+SimulationManager.getLastSimulation().getVVal().toFormattedString());
```

6. Print out data from the sensors
    6.1. Get the sensor list from the most recent simulation

```
System.out.println("Current Sensors:"); System.out.println("");
for (Sensor s : SimulationManager.getLastSimulation().getSensorList()) {
       System.out.println("Sensor Number: "+s.getID());
       System.out.println("Alpha: "+s.getAlpha().toFormattedString());
       System.out.println("H-Value: "+s.getHVal().toFormattedString());
       System.out.println("N-Value: "+s.getNVal().toFormattedString());
       System.out.println("");
}
```

7. Sort the sensors
    7.1. Use the SimulationManager to adjust this

```
SimulationManager.sortSensors();
SimulationManager.sortSensors(0);
SimulationManager.sortSensorsByNumber(true);
SimulationManager.sortSensorsByAlpha(true);
SimulationManager.sortSensorsByH(true);
SimulationManager.sortSensorsByN(true);
```

8. Use the SetupListener
    8.1. Implement SetupListener in your class, then call *SimulationManager.setSetupListener(this);*