GCM 3.2 Modeling Guide

# Chapter 1: Introduction

## Who is this for?

The General Computation Model (GCM) is a Java based simulation framework for building disease progression models. Users of GCM should have a general familiarity with Java and object oriented programming and would benefit from some exposure to event based modeling.

## High level overview

### Simulation

GCM is an event based simulation composed of data managers, actors and an event engine. The data managers contain the state of the simulation and generate events when that state changes. The actors contain the business logic your model and act on the data managers. The engine transports events generated by the data managers to any data managers and actors that subscribe to those events.

### Plugins

Data managers and actors are organized into plugins. A GCM model is thus composed of the core simulation and a suite of plugins. The plugin architecture provides for the scalable reuse of concepts and capabilities between models. GCM is provided with a set of existing plugins that define many of the concepts useful to a broad range of models such as the management of people, their properties, social group structures and the like. The modeler is free to compose a model from their choice of plugins.

### Experiment

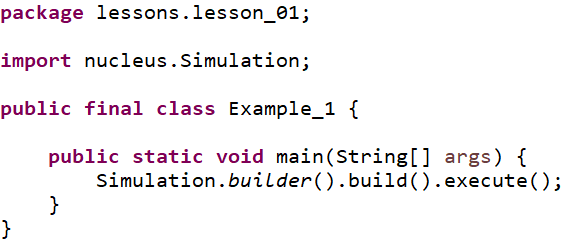
GCM also provides an experiment management system. Each plugin contains zero to many data objects that define the initial state of its actors and data managers. Each such data object may be altered freely. The complete set of all combinations (scenarios) of the variant plugin data objects form an experiment and a separate simulation instance is executed for each combination.

# Chapter 2: Getting Started

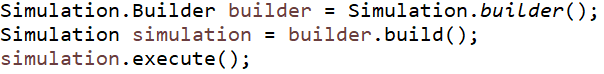
We start with a set of practical lessons that will help clarify the core concepts of GCM. The lessons generally build on one another and should be taken in order. You are encouraged to code along with the lessons.

## Lesson 1: Hello World

Our first lesson is a very reduced “Hello World” example where we will execute the simulation with one line of code.



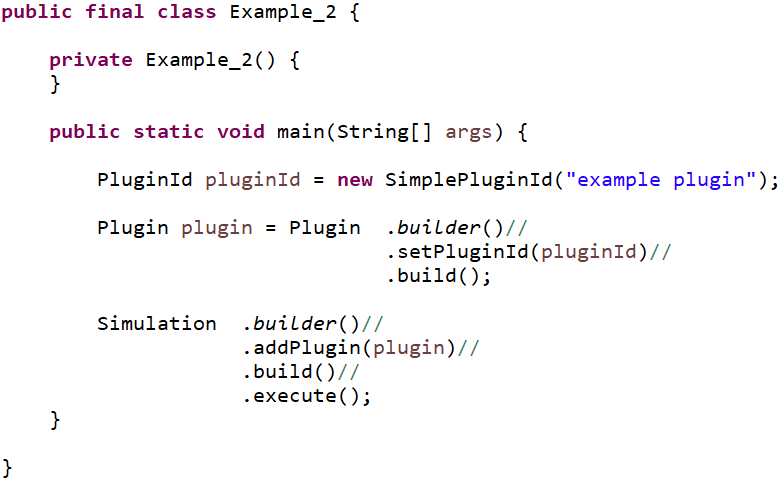
With this one line we have created and executed a simulation. Since the simulation had no actors or data managers there was nothing to do and so it terminated immediately. Let’s analyze the line in a more draw out form:



The simulation does not have a constructor. Instead it uses a static builder class that creates the simulation from various arguments. The builder is immediately capable of building a simulation instance so we will skip giving it any more information. The simulation is only capable of executing, so we execute it.

## Lesson 2: Add a plugin

Models are made of plugins. In this lesson we will add a single plugin to the simulation and execute it.



The first thing we will need to do to build a plugin is to identify it. The PluginId is a marker interface – it has no methods and serves to help differentiate between plugin id values and other identifiers. The SimplePluginId is a convenience implementor of PluginId and will wrap any object as an identifier. In this case we use the string “example plugin”, but you are free to implement them however best fits your needs.

Next we build the plugin. The Plugin class implements all plugins and you can provide several arguments to its builder to specify the contents and behavior of your plugin. A plugin is composed of four items:

1. An id
2. Dependencies on other plugins
3. Data objects used to initialize data managers and actors
4. An initializer to load the data into the simulation

For now, we will only need to add the plugin id and build the plugin.

Finally, we build the simulation by adding the plugin and then executing as usual. The result is the same as the previous lesson: nothing happens. However, internally, the simulation did add the plugin and found it had no information other than its id.

## Lesson 3: Adding an actor to the plugin

### Contexts

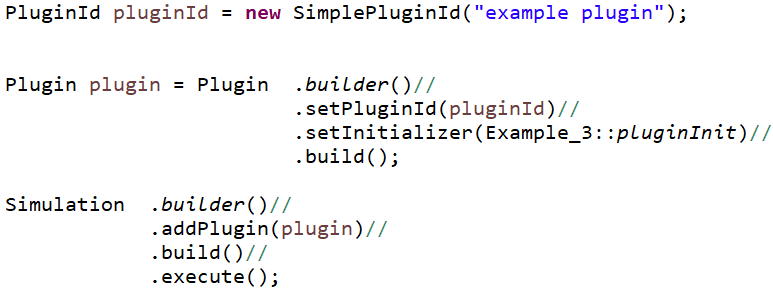
In all that follows, we will encounter various context objects. Contexts are interfaces into the simulation that are tailored to the thing using the context. For example, an ActorContext provides everything that an actor will need to interact with the simulation. Similarly, a DataManager context provides the capabilities needed by data managers.

The first context we encounter is the PluginContext. It provides the plugin with the following abilities:

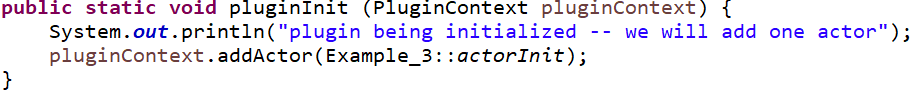
1. Add an actor to the simulation
2. Add a data manager to the simulation
3. Get plugin data

The PluginContext is passed to the plugin’s initializer and is used to add all data managers, all initial data and any actors that need to exist at the beginning of the simulation run.

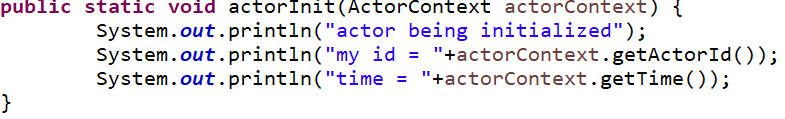
The next context will be the ActorContext. It provides actors with a wide array of capabilities that we demonstrate later. For now, the important takeaway is that being granted a context implicitly identifies the recipient as having a particular role in the simulation.



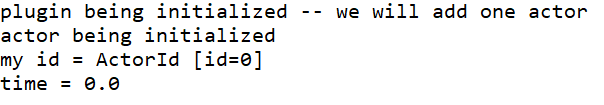
We are setting the plugin’s initializer. The initializer is a method that consumes a PluginContext and returns void. For this example, we use a static local method for our initializer:



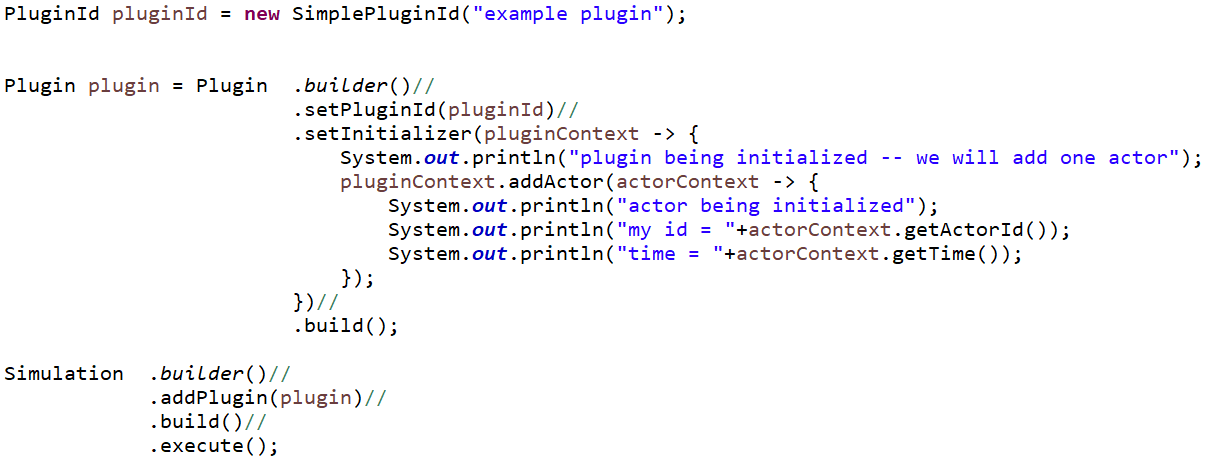
When the simulation starts up its execution, one of the first things it will do is to execute each plugin’s initializer to give the plugin an opportunity to add actors and data managers to the simulation before time and events begin to flow. Adding an actor is done with another consumer, but this time it is a consumer of ActorContext.



After the plugins are initialized, the actors and data managers are next. For this example, the actor is initialized and it prints a few statements and ceases activity. Here is the resulting console output:



We can replace the local method references above with lamdas to be more succinct.



## Lesson 4: Adding a data manager to the plugin

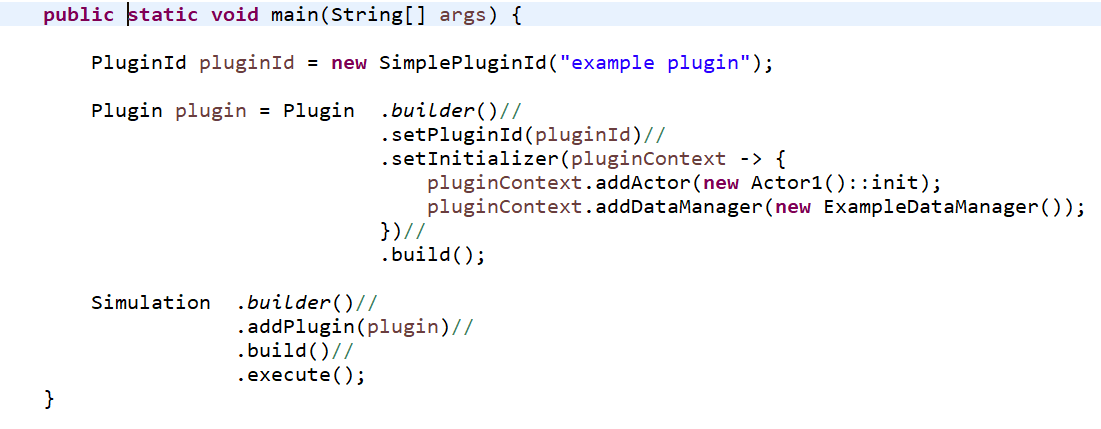
We extend the previous lesson by slightly altering the actor and adding a data manager. But first let’s list some of the attributes of data managers and actors to better understand the roles they play in the simulation.

Data Managers

* Exist for the full duration of the simulation
* Contain and maintain the entire state of the world.
* Are highly stateful
* Produce events in reaction to state changes
* Interact with other data managers via events
* Do not have a set of objectives. They are not trying to achieve some particular state of the world
* Are narrowly focused on some particular aspect of the world, but are concerned with all instances of that aspect
* Are added as instances and are limited to a single instance per class type

Actors

* May be added and removed over time
* Are not considered to be part of the world
* Are generally stateless
* React to but do not produce events
* May access any data manager
* Have objectives. They contain the business logic of the model and are trying to achieve some particular state of the world
* Are concerned with many aspects of the world, but often focused on a particular subset of world
* Are added as consumers of ActorContext and may be composed of any such consumers



We add an instance of ExampleDataManager to simulation. Unlike the actor, where we pass a consumer of context, we need to provide an actual instance of a data manager. Note that the ExampleDataManager extends the base class DataManager. The base class provides the only init() method to override and you must include the super.init(dataManagerContext) call as its first line. This is done to ensure that each data manager is initialized exactly once by the simulation.

The ExampleDataManager has two (completely arbitrary) data fields alpha and beta and provides both getters and setters for each.



The actor is now specified via the ExampleActor class. Most actors contain enough code that we usually put that code into a separate class rather than a lambda statement as we did in the previous lesson. Note that the init() method has the correct method signature of being a consumer of ActorContext.

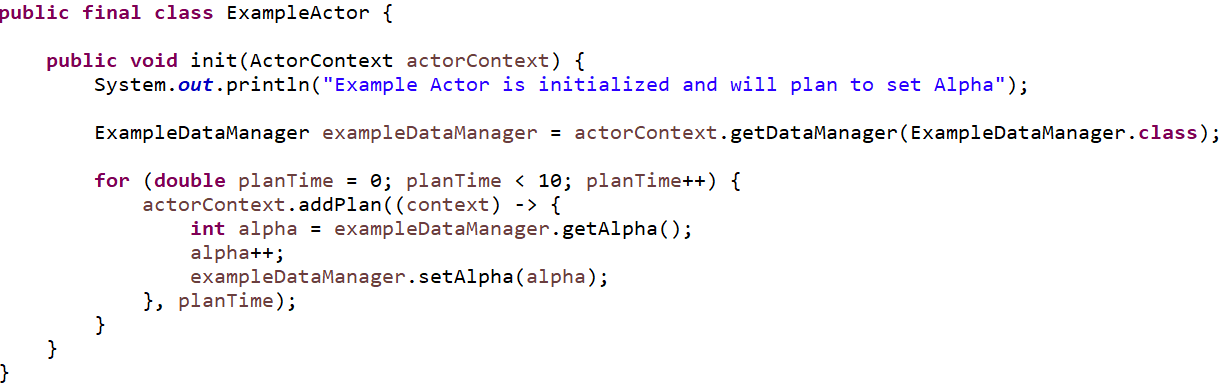
### Plans

In GCM, an actor can do three things:

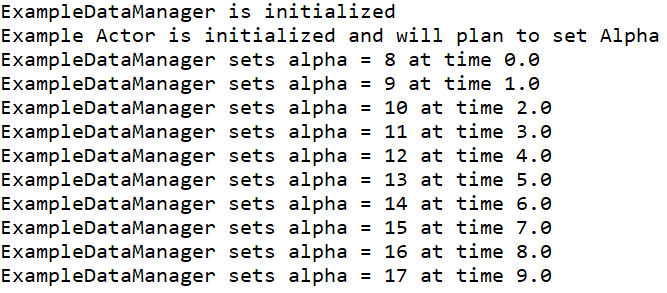
1. **Observe**: Observation can be done directly by gaining access to a data manager and then getting a value from that data manager. Observation can be done indirectly by subscribing to events. We will cover that option later.
2. **Act**: A mutation to some data manager’s managed data.
3. **Plan**: At some time in the future, the actor will take some particular action

Actions in GCM are always executed in the current moment in the simulation. Unlike many future event simulations where events are queued for future execution, GCM allows an actor to plan for an action or observation in the future. The plan is a consumer of ActorContext and can be a static method, member method or a lambda. The plan is registered with the simulation and is executed only when time has moved forward to the plan’s scheduled time. There is no requirement that the plan do anything at all. This allows the flexibility to re-evaluate the circumstances of the planned action and choose to take appropriate action at that time. Plans are queued in GCM by their associated planning times and it is this queue that dictates the flow of time. For example, suppose the simulation finds the first plan is scheduled for time= 2.4 days. The current time = 0 days and the simulation progresses time to 2.4 days and then invokes the plan. Plans are always privately managed by the actor that owns the plan and no other actor or data manager has any insight into those plans. See the planning chapter for more details on planning.

In this example, the actor is initialized at time= 0 and generates 10 plans to increment the value of the alpha in the ExampleManager. Each time the ExampleManager changes the value of alpha, it outputs to the console a description of the change.

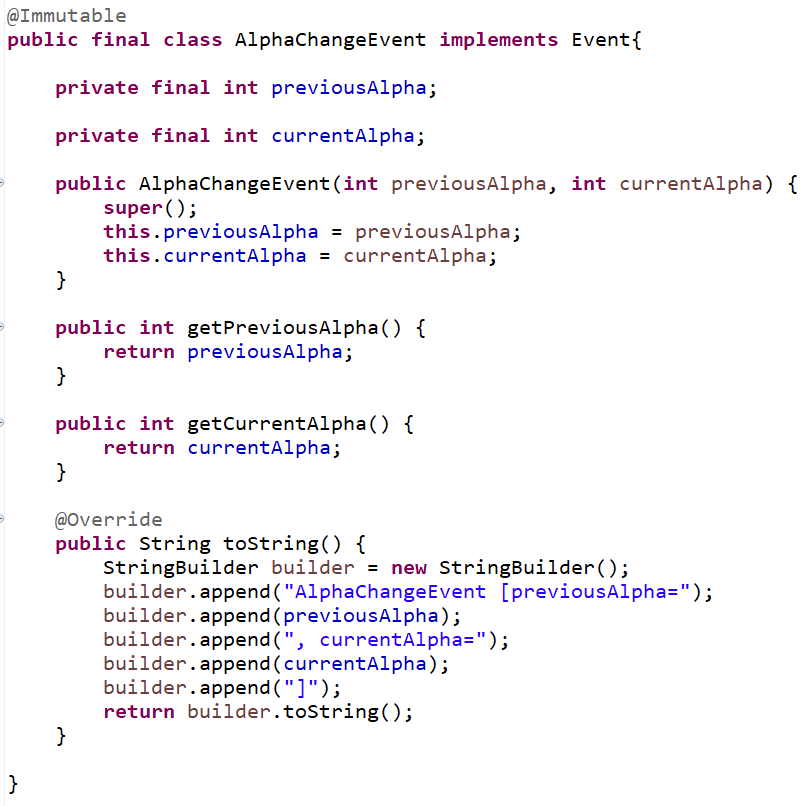


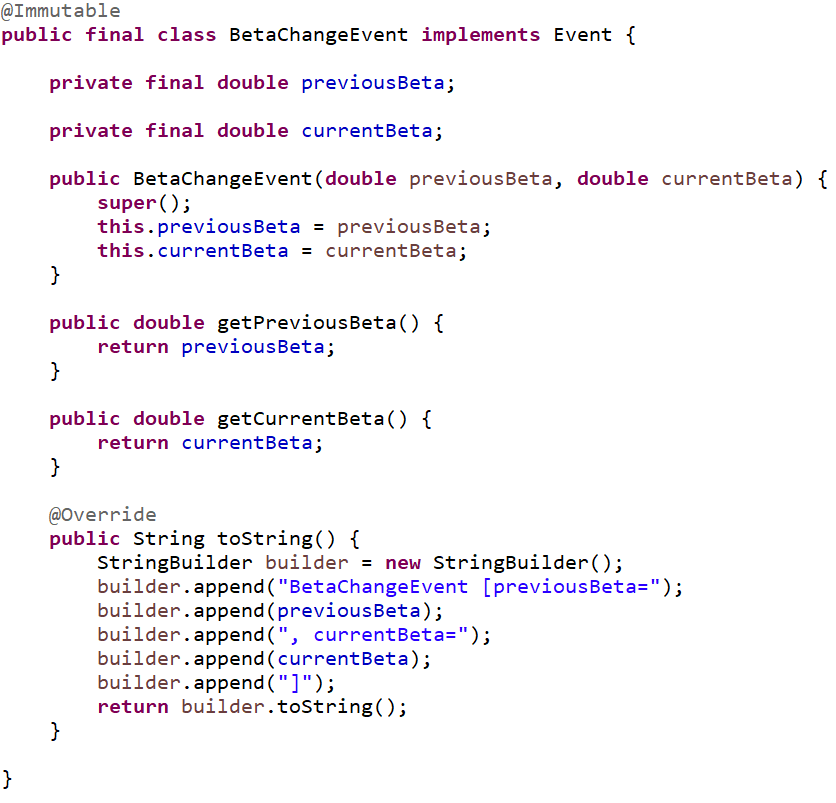
The output from the simulation is:



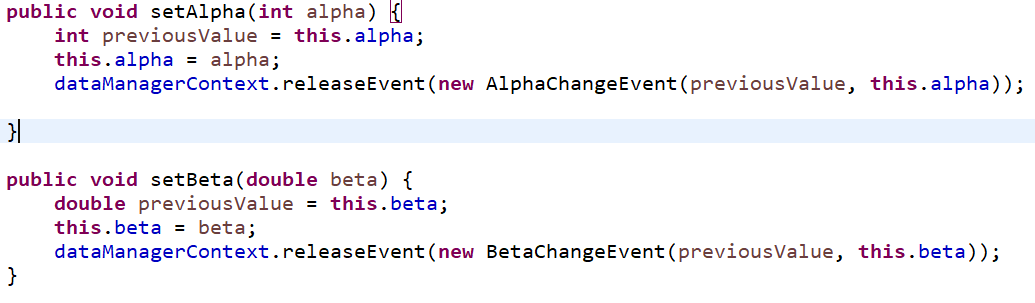
## Lesson 5: Introducing Events

An event in GCM is a notification of a data change to the stage of a data manager. In this example we will introduce two events corresponding to the two changes to the ExampleDataManager. Both events document the previous value and current value (at the time when the event was generated) and are immutable data classes.



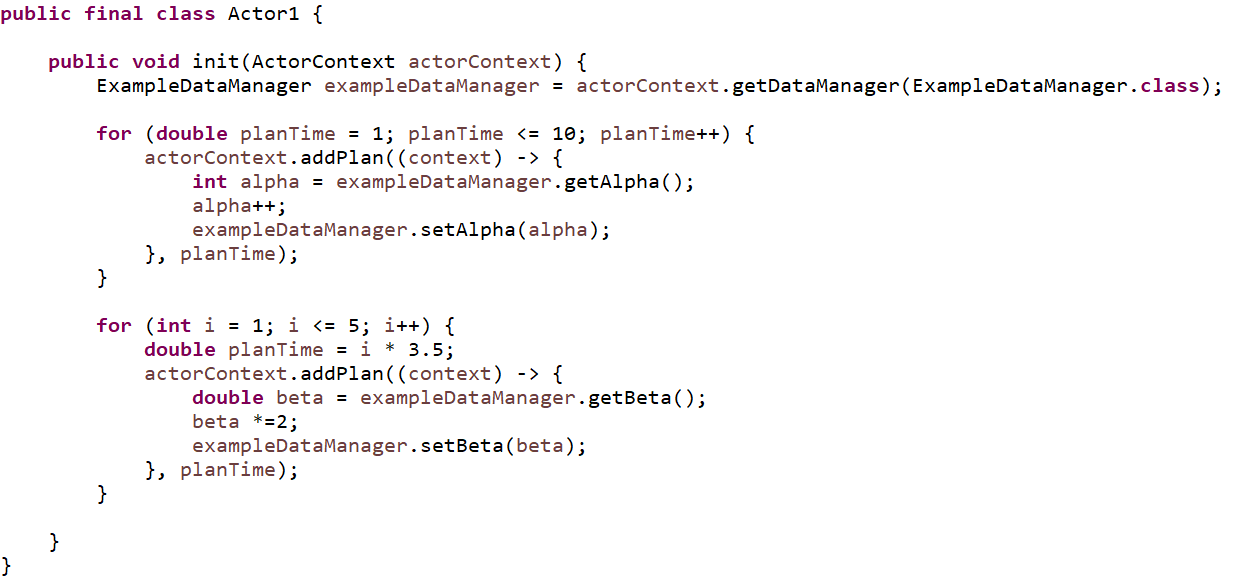


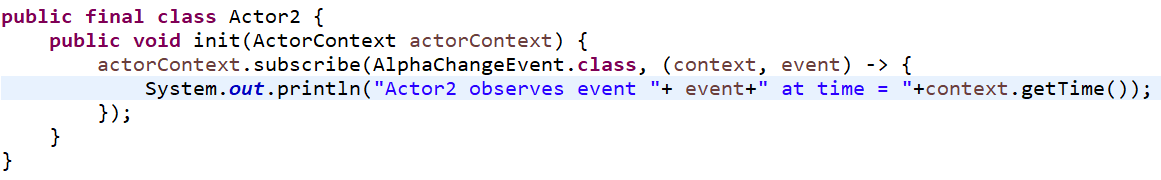
Each is generated by the ExampleDataManager when the alpha or beta values are mutated by releasing the events through the DataManagerContext to the simulation:

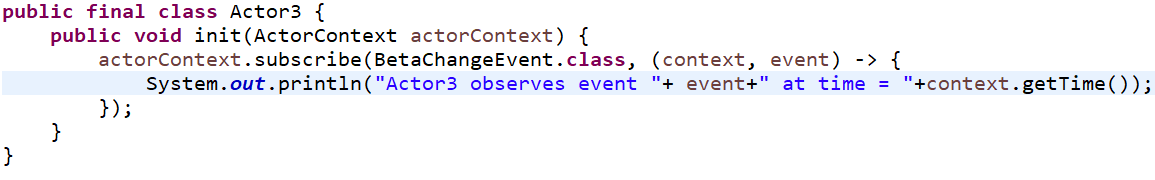


There are three actors in this example:

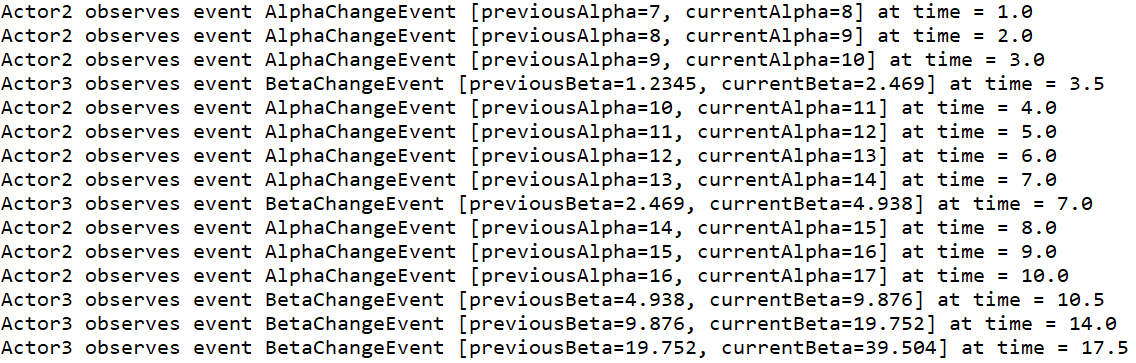
1. Actor1 makes changes to both the alpha and beta values at 1 and 3.5 day intervals respectively
2. Actor2 subscribes to AlphaChangeEvent events and reports to console what it receives
3. Actor3 does the same for BetaChangeEvent events







The resulting console output shows Actor2 and Actor3 observing the expected events at the expected times:



## Lesson 6: Introducing plugin dependencies

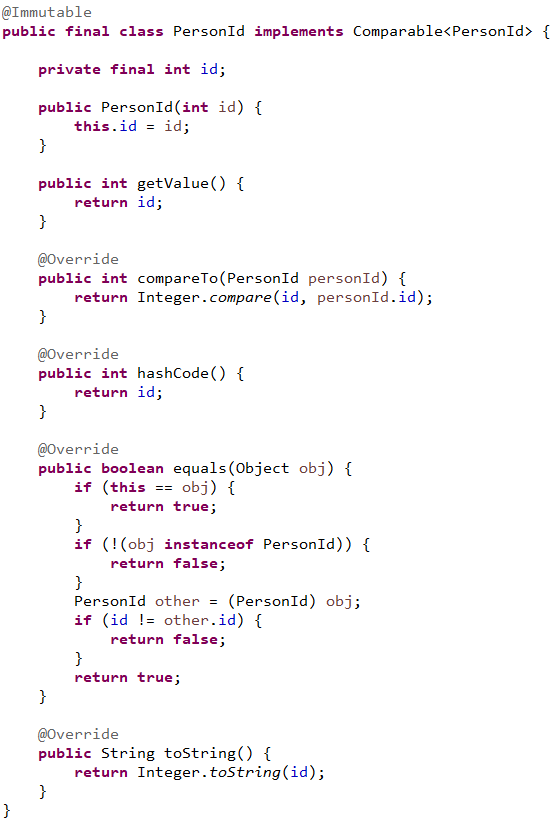
So far we have covered what actors and data managers do and that they are introduced into the simulation via plugins. Over the next lessons we take a closer look at the plugins. This lesson starts with creating a more realistic set of plugins each arranged into separate java packages.

* People plugin
  + Defines a person id
  + Adds the PersonDataManager for tracking people
  + Adds events for the the addition and removal of people
* Family Plugin
  + Defines a family id
  + Adds the FamilyDataManager for grouping people into families
* Vaccine Plugin
  + Adds the VaccineDataManager for tracking which people have been vaccinated
* Model Plugin
  + Contains the ModelActor class to add people organized into family structures and vaccinate some of those people

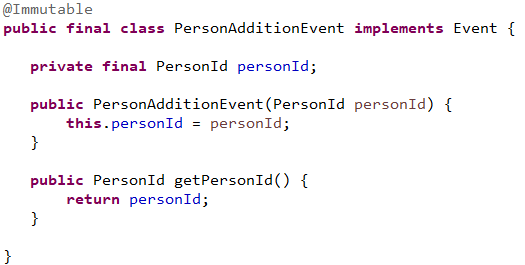
Here are the classes that implement this example:

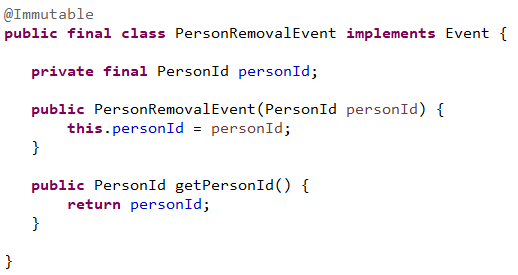
### People Plugin:

The people plugin defines a PersonId as a simple, immutable wrapper to an int value. The PersonDataManager tracks people via PersonId values and allows for the addition and removal of people. PersonId values are generated in order and never reused. Events are generated when people are added or removed.



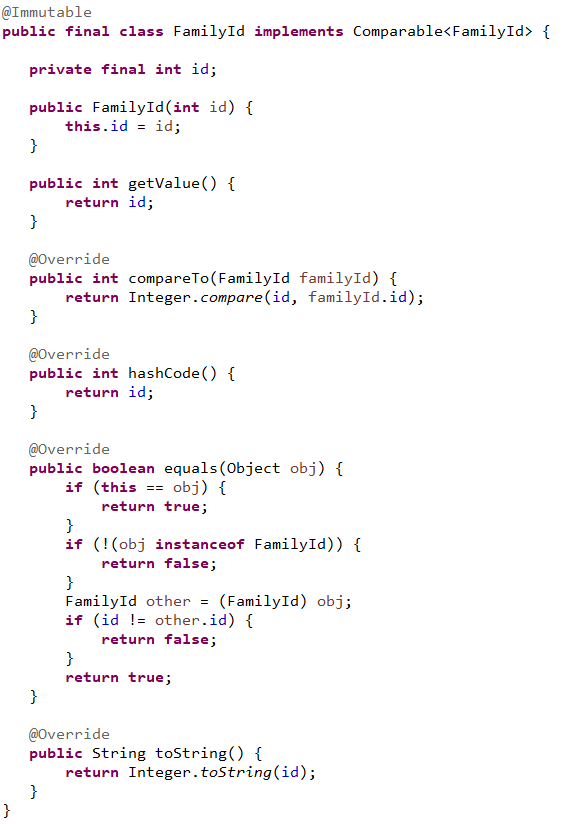


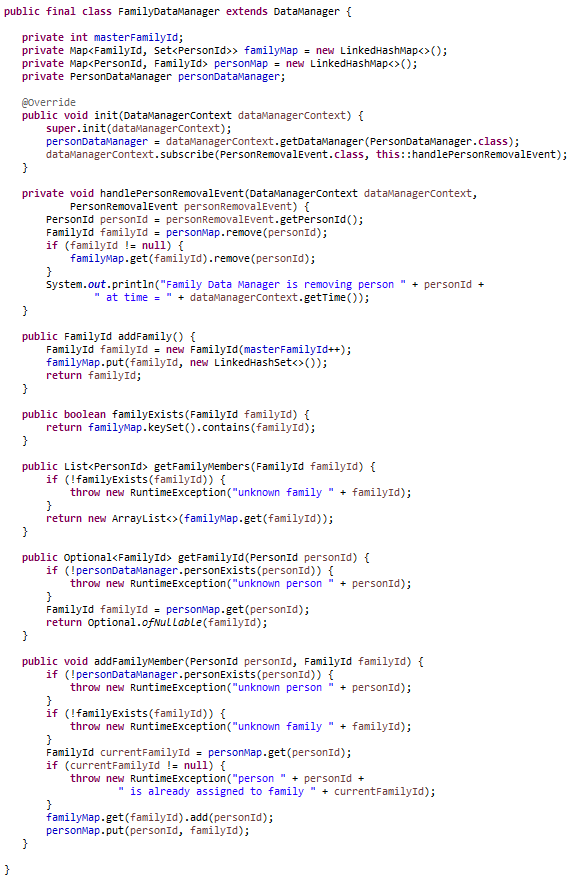




### Family Plugin

The family plugin defines a FamilyId as a simple, immutable wrapper to an int value. The FamilyDataManager tracks family membership via two-way mappings of PersonId to FamilyId. In this example families can only be added and people can only be added to families. However, people can be removed via the PeoplePlugin so the FamilyDataManager subscribes for PersonRemovalEvent and thus removes the people from families.





### Vaccine Plugin

The vaccine plugin contains only the VaccineDataManager which tracks by PersonId which people have been vaccinated. Like the FamilyDataManager, it too subscribes to PersonRemovalEvents and adjusts its data accordingly.



### Model Plugin

The model plugin contains a single actor, the ModelActor, that serves to:

* Add people to the simulation
* Group them into families
* Vaccinate some people
* Demonstrate that events cascade

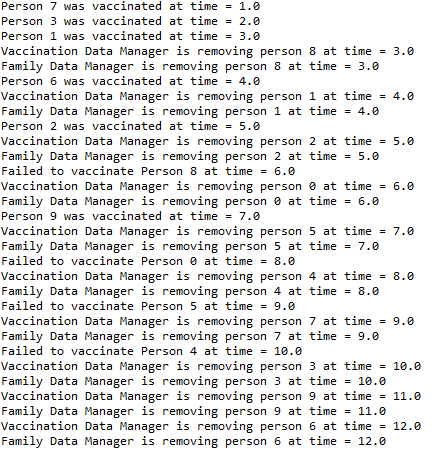
### Connecting the Plugins

Both the family and vaccine plugins depend on the concept of a person as implemented by the PersonId class. They also need to respond when a person is removed from the simulation and do so by handling the corresponding PersonRemovalEvent generated by the person plugin. We build these dependencies via the Plugin.Builder class in the example code below.



Note the addition of the dependency on the people plugin via its id when adding both the vaccine and family plugins. The order of addition of the plugins to the simulation is relatively unimportant as is ordering in general in any of the builder patterns used in GCM.

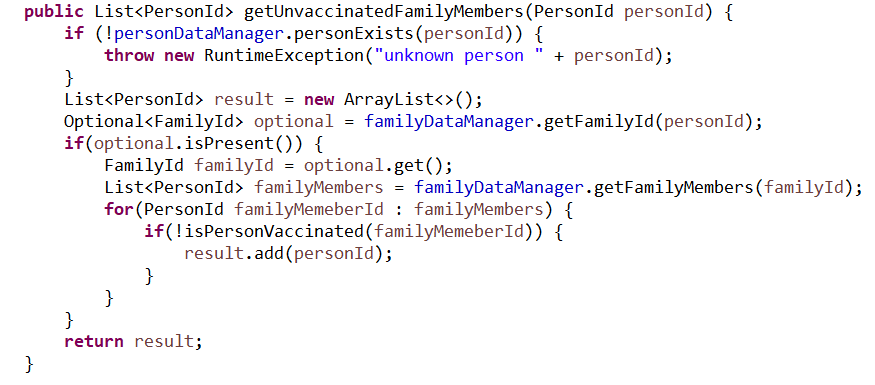
The resulting output:



## Lesson 7: Plugin dependencies continued

We extend the previous lesson by adding an additional dependency of the vaccine plugin on the family plugin. This will allow the VaccineDataManager to answer queries about which members of a family have yet to be vaccinated.

From the VaccineDataManager:



The plugins in this example form a dependency pattern:

people

family

vaccine

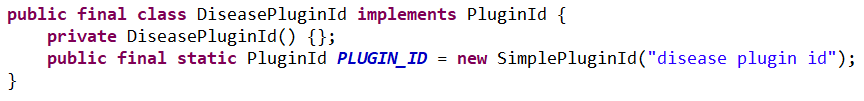
All plugin dependencies in GCM form similar directed, acyclic graphs (DAGs). There can be no loops in the dependency graph, but the graph does not have to be fully connected. The dependencies reflect the requirements of the data managers within a plugin to access data managers in other plugins. This pattern drives the order in which events are presented to data managers.

In this lesson, the VaccineDataManager and the FamilyDataManager have both subscribed to the PersonRemovalEvent generated by the PersonDataManager. Since the VaccineDataManager also has a dependency on the FamilyDataManager, the VaccineDataManager should receive the event after the FamilyDataManager. Events cascade through the subscribed data managers in an order that is consistent with the plugin dependency DAG.

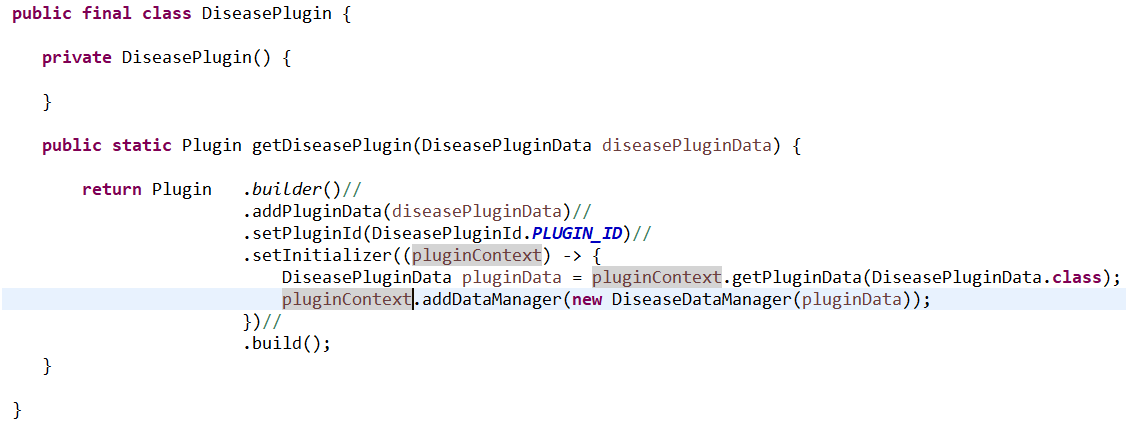
## Lesson 8: Plugin Packages and Plugin Data

The Example code in the last lesson was a bit verbose and can be improved. Identifying and generating the plugins can be included in the plugin packages by introducing classes for each id and classes for each plugin’s contents.

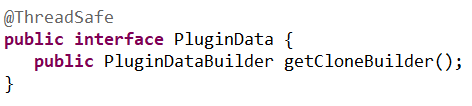
In the disease package we add a unique plugin identifier with a final static id field:



We also add a static class (DiseasePlugin) that implements the construction of the plugin from the required plugin data.



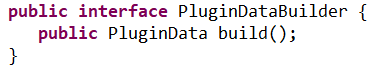
The plugin is initialized with a DiseasePluginData object that contains the initial values for r0, asymptomatic days and symptomatic days. Most plugins will have a single plugin data object, but some may not need any and some may be designed with multiple such classes. All such classes must implement the PluginData interface:



Plugin data classes must be threadsafe since they will be shared between multiple simulations running on separate threads. This stands in contrast to the actors and data managers which are created and managed in the thread of a single simulation. The best practice is to make plugin data classes immutable since immutable classes in Java are guaranteed to be threadsafe. For a class to be immutable in Java it must meet three conditions:

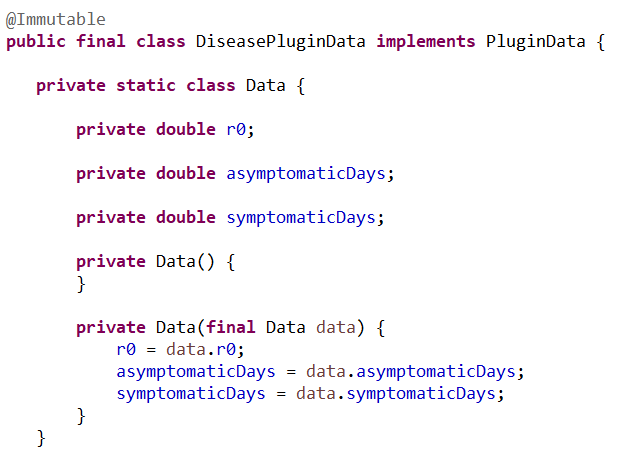
1. It cannot be mutated, i.e. it has no setters and no public fields.
2. All its fields are marked final.
3. Its constructor(s) do not pass reference to self. No reference to the newly created object leaks out before construction is complete.

Besides carrying whatever data is needed by the plugin, the PluginData implementor must provide a PluginDataBuilder:



The role of the plugin data builder will be explored in the next lesson where it will be used to make alterable copies of plugin data to drive the experiment. For now, let’s examine the DiseasePluginData class. It is composed several sections:

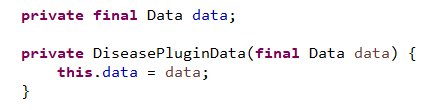
* A data class
* A static builder class
* A single data field and private constructor
* Getter methods for the data
* A clone builder method



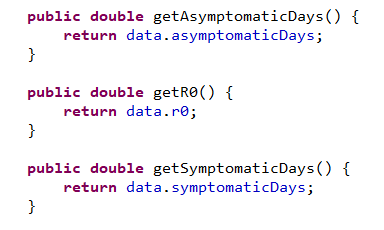
The Data class is private and just contains the fields needed by the plugin. Note that it is a mutable class and that its fields are not final. It will be used by the builder class later to store values. Its constructors are private and allow one Data object to be copied from another.



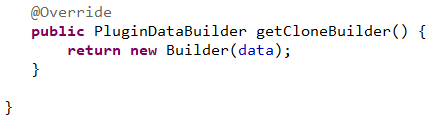
The static builder class is used instead of a constructor. The use of builder classes for plugin data objects is key to the creation of experiments covered in the next lesson. For now, let’s concentrate on what the builder does. First, it has setter methods for each of the data fields and each such method returns the builder instance to support method chaining. Next, the build() method returns the DiseasePluginData via a try finally block so that if anything goes wrong, the builder instance will uncorrupted and ready for reuse. Note that after invoking the build method, the data object gets replaced with a fresh version. Finally, the builder’s own constructor is private and is accessed via a static method. This is done to grant a syntax that is more compatible with the method chaining. We will defer discussion of the private dataIsMutable field until the end of this section.



After the builder collects the data, it passes that data to the instance of the DiseasePluginData which is stored as a final field. Recall that the field must be final in an immutable class.



The getter methods for each field value in the data are added. There are no corresponding setter methods.



We end the class with the getCloneBuilder method.

*Our use of the term clone is intuitive but may cause some confusion. What we are doing is copying the data in the DiseasePluginData and placing into a builder so that it can be further mutated later in the experiment. Java formally defines the term clone as a part of the Object class definition and implements it with a protected method clone(). Use of the Object.clone() method has generally fallen out of favor in Java but still has some proponents/use cases.*

The method returns a new Builder that has reference to the current data object. The builder starts out with the private field **dataIsMutable** set to false. This is to ensure that we do not pay the cost of copying the data object if we never invoke any of the setter methods before invoking **build()**; While such considerations seem trivial with only three values, in most models there will be plugin data classes that contain millions of data values.

The resulting example class is easier to read and more succinct:

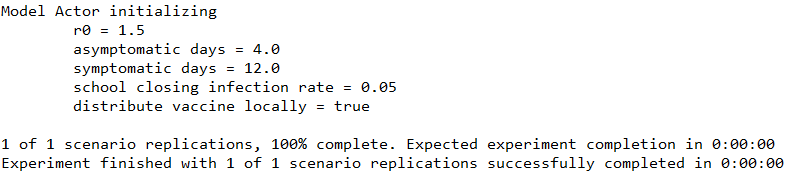


## Lesson 9: Introducing Experiments

So far we have mentioned that the plugin data classes play a role in executing an experiment via the **getCloneBuilder** method. Let’s start with the simple experiment. We will update the last example class by replacing the Simulation execution with an Experiment execution:



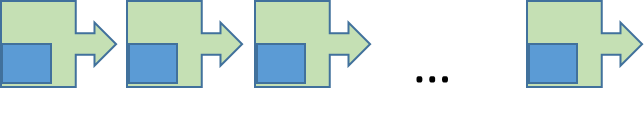
The experiment class has a very similar builder to the Simulation class so we only have to swap out the Simulation reference for an Experiment reference. The resulting execution created an experiment containing exactly one simulation that runs in the main thread. However, the output contains information about the status of the experiment.



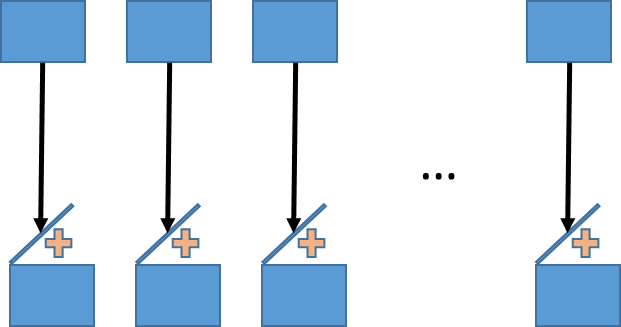
### What happens when the experiment executes?

You have contributed several plugins to the experiment and on execution the experiment generates multiple simulation runs on multiple threads. Let’s examine how this is accomplished as a way to motivate this lesson’s code examples.

The experiment is composed of several plugins, each with zero to many plugin data objects. For purposes of the diagrams we will assume that each plugin has a single plugin data object.

**plugins**

The experiment gathers the plugin data objects and gets the plugin data builder for each. These plugin data builders will come pre-filled with the data from the original data objects.



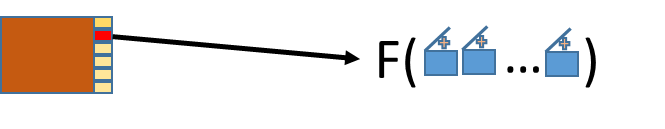
By altering the data in these builders, we generate new scenarios for the simulations to execute. GCM manages the instructions to alter the plugin data via Dimensions. Each dimension contains one to many levels.



For example, we may have a dimension that alters the value of **alpha** from **plugin data A** and the value of **beta** from **plugin data B**. Each level in the dimension will set specific values for alpha and beta via the builders.

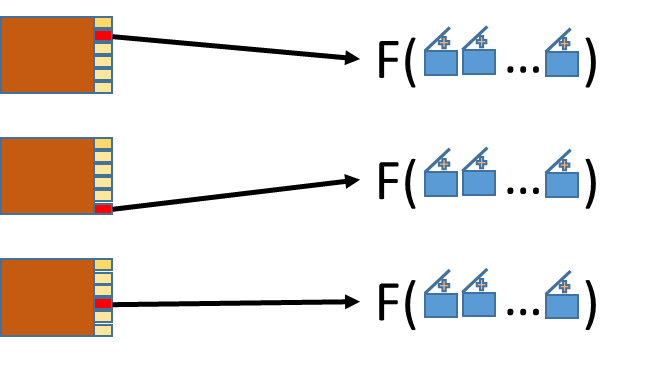
|  |  |  |
| --- | --- | --- |
| level | alpha | beta |
| 0 | 2.3 | FALSE |
| 1 | 3.6 | TRUE |
| 2 | 4.8 | FALSE |

Each level in a dimension is actually a function that takes in the builders and manipulates the content of each plugin as needed.

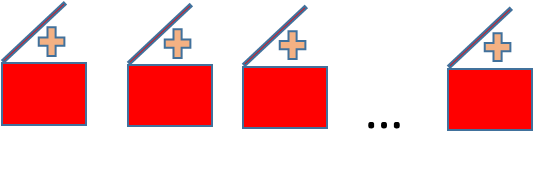


Consider an experiment with two dimensions having 3 and 5 levels respectively. The number of level permutations is 3x5 = 15. Each such permutation is referred to as a scenario and the scenarios are numbered from 0 to 14.

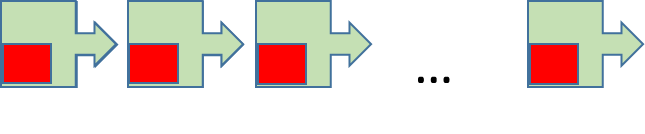
As the experiment executes, it works with each scenario id and determines for that id which levels are active for each dimension.



Each level (via its function) alters the contents of the builders in turn, resulting in a unique set of content for that scenario.



The builders are then instructed by the experiment to build the plugin data objects. The resulting data objects are inserted into copies of the original plugins to produce a unique set of altered plugins that are specific to the scenario id and executed via a single simulation instance.



You may have noticed that the initializer code above acquires the DiseasePluginData via the context rather than the instance passed to the getDiseasePlugin() method. This is a necessity due to experiment design and will be covered in the lessons that follow. **In general, the initializer code should always retrieve plugin data from the plugin context.**

We expand the example by adding a single dimension that set r0 to two values, generating two simulations.

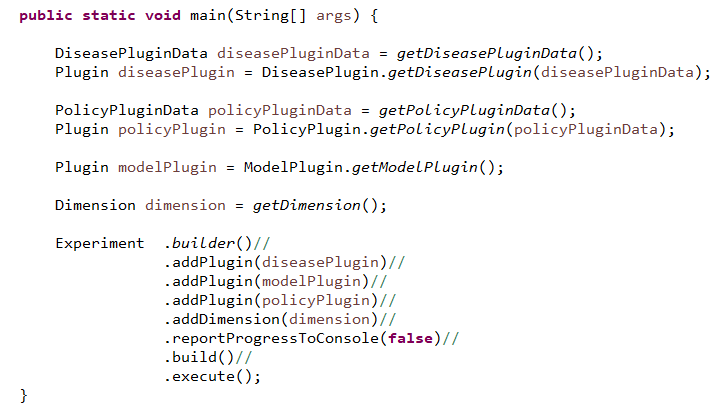


In the dimension we see that there are two levels and the addition of some meta data in the addMetaDatum(“r0”) invocation. The meta data here represents the information that each level is altering in the experiment. The main purpose of each level is to alter the state of a builder(s) but must also return meta data values to match the meta data for the dimension. The meta data of the dimension acts as a header to a table while the meta data for each level are the values in that table.

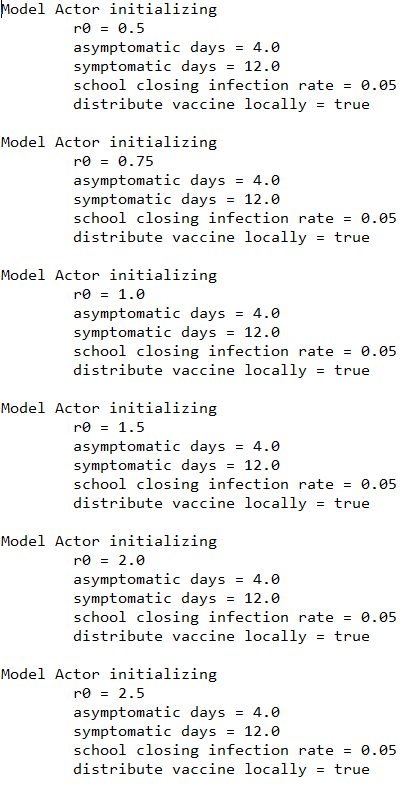
The building of the dimension can be streamlined without typing out each level:



The resulting experiment execution is more streamlined:



We have turned off the experiment report progress to console in the code above. We have chosen six values for r0 in our dimension and thus we have 6 simulation executions, each having the model actor print out the contents of the DiseaseDataManager:



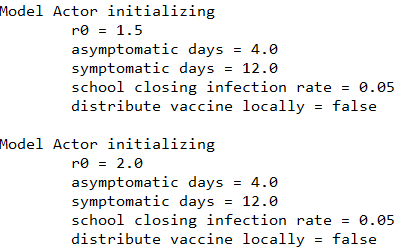
We extending the example again, reducing the r0 dimension to just three levels and introducing a dimension over the policy data. This new dimension has four levels controlling local vaccine distribution and school closing infection rates:



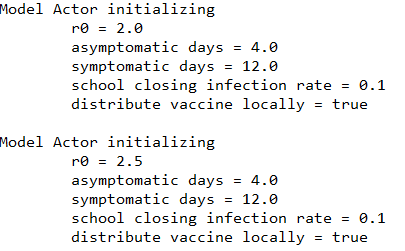
We add the new dimension to the experiment:



The result is now 12 executed scenarios:



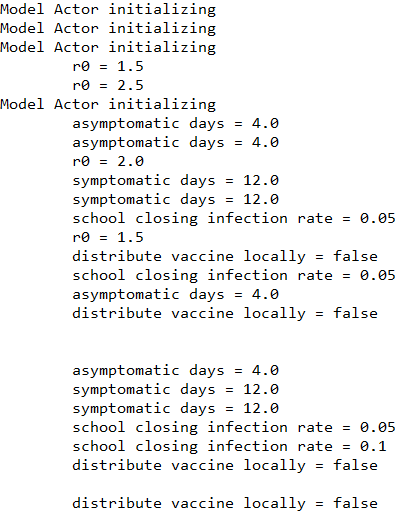
…



So far, the experiment has run in a single thread. We now run it in four threads by adding a single line to the experiment giving is a thread count:



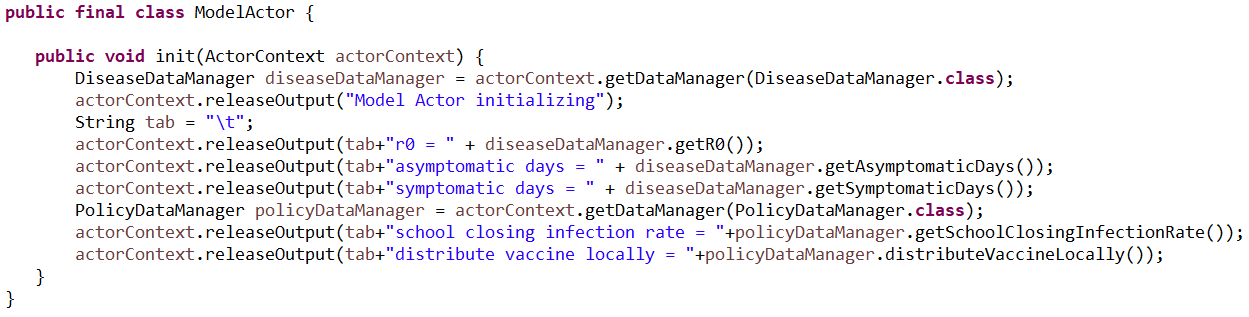
The experiment runs in the main thread and the scenarios now run the four additional threads. The resulting console output a bit jumbled since the writes to the console are now coming from four simultaneous simulation runs:



We will alleviate this problem as we explore how the simulation and experiment manage output.

## Lesson 10: Output

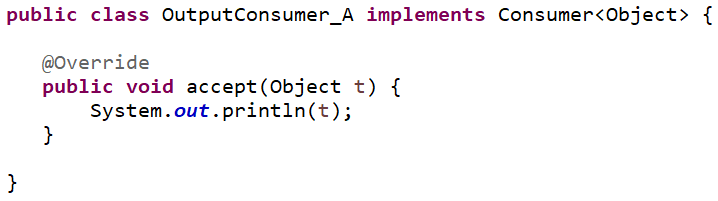
So far we have only produced output by writing directly to the console in the various actors and data managers. The simulation contexts (ActorContext/DataManagerContext) provide for the release of output objects to an external handler (outside the simulation). In this lesson, the ModelActor class has been altered to use this mechanism:



Data managers can release output in a completely similar way. The output objects are handled by an external handler presented during the build of the simulation:



Released output objects are sent to the output consumer. In the current example, that consumer is an instance of the class OutputConsumer\_A and it simply prints the object to the console:



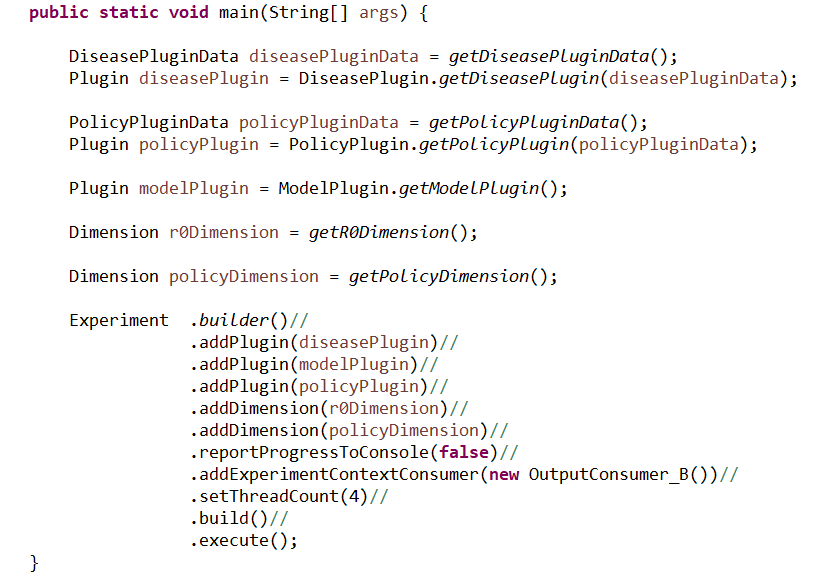
At first glance this mechanism seems simple and not particularly useful. In practice, one rarely uses the simulation directly and instead favors the experiment which has a somewhat more sophisticated handling of output. With experiments, GCM is potentially using multiple threads to execute each simulation, so output handling must be threadsafe.

### Experiment Context

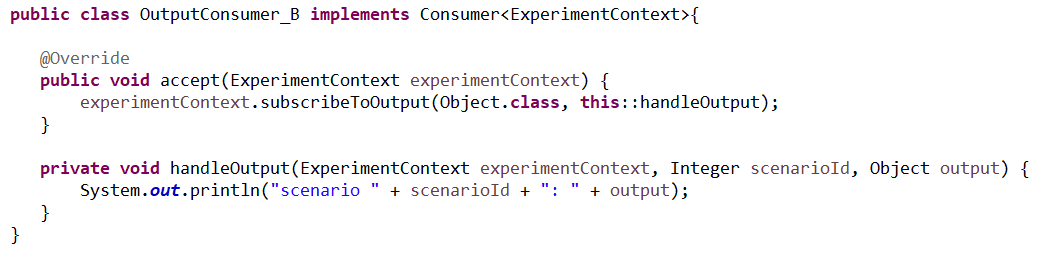
Just as the simulation supplies contexts, the experiment uses the ExperimentContext to give output consumers a view into the ongoing experiment. It gives each output consumer several capabilities:

* Subscription to output by output class type
* Subscription to the opening and closing of the experiment
* Subscription to the opening and closing of each simulation
* Scenario status information
* Experiment and Scenario meta data

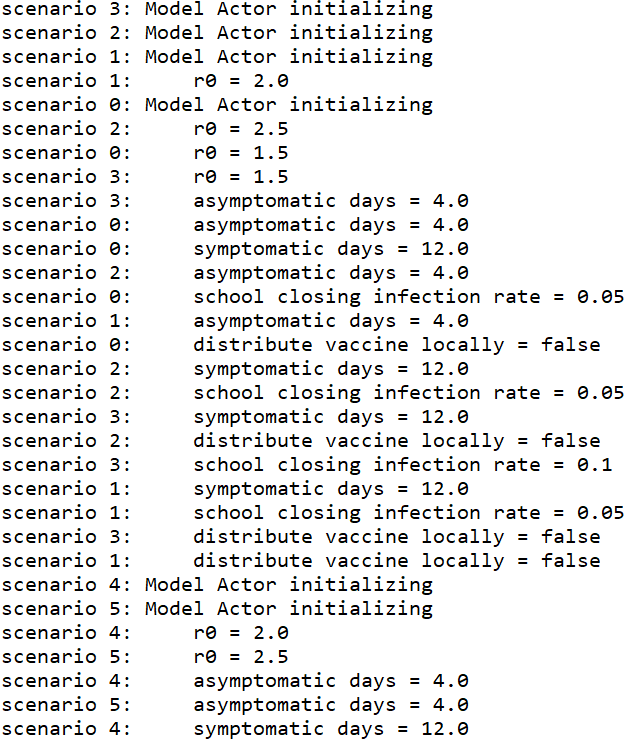
In Example\_10\_B, we bring back the dimensions from previous lessons and will excerpt just the main method:



Like the simulation, the experiment is adding a consumer for output, but this time that consumer is “consuming” an experiment context. Once the consumer receives that context, it will use it to further subscribe to output and various experiment level events.



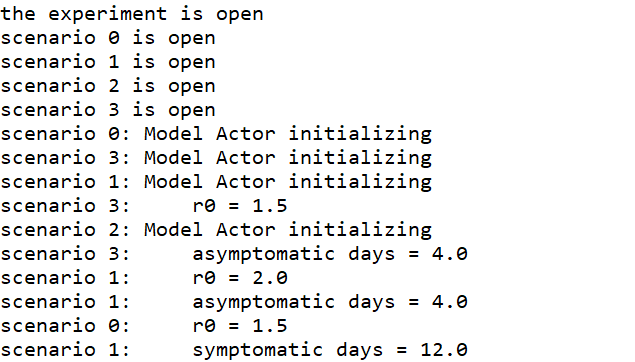
The experiment can have any number of ExperimentContext consumers and initializes each at the beginning of its execution via the accept() method. In OuputConsumer\_B, the only action the consumer takes is to subscribe to all output and have that output handled by the handleOutput() method. The resulting output shows the scenario id for each line:



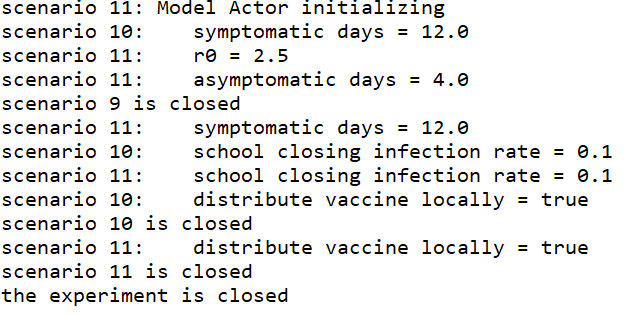
Example\_10\_C switches the experiment context consumer to an instance of OuputConsumer\_C which subscribes to all output types as well as the opening and closing of the experiment and all simulations (scenarios):



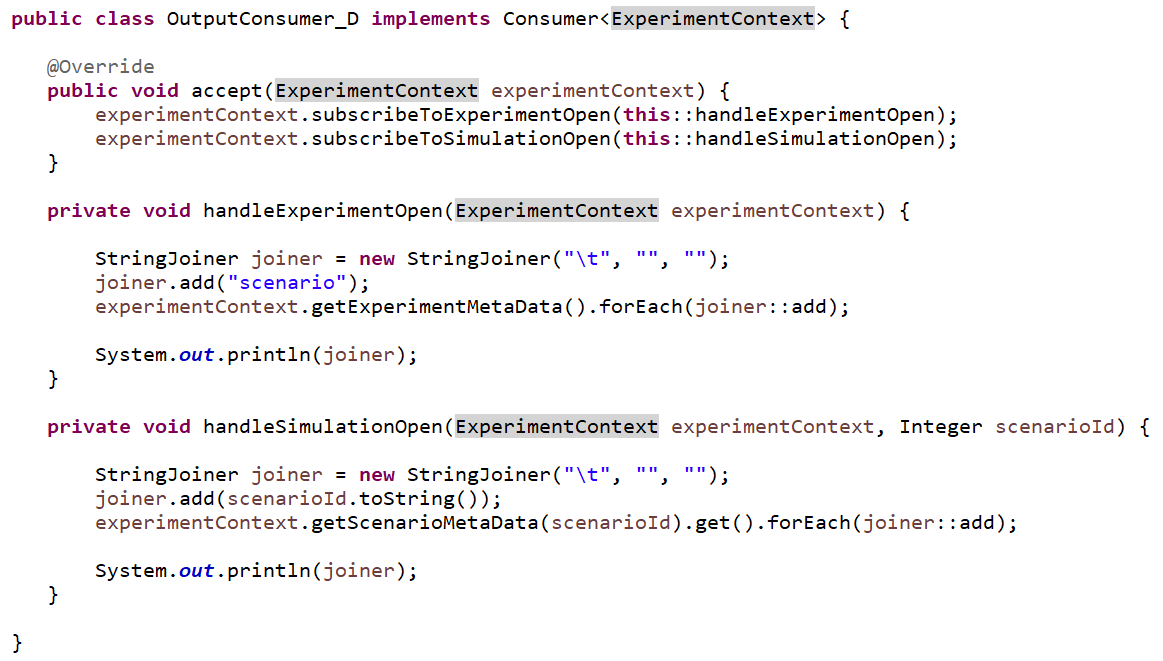
The resulting output shows the usual released output along with the opening and closing of each simulation:



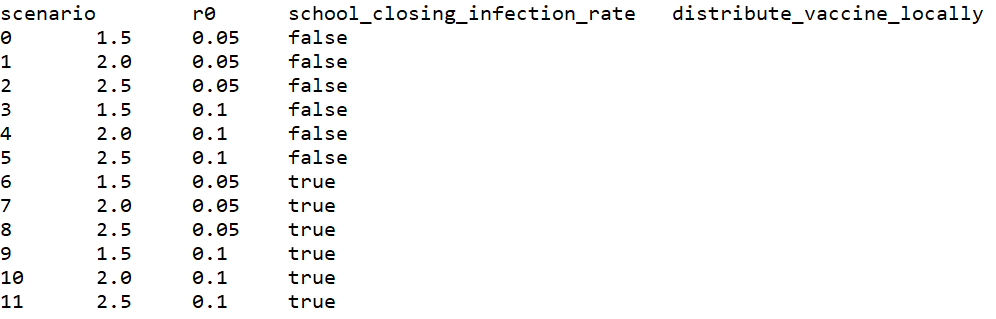
…



In the final example, OuputConsumer\_D, we drop the output handling and demonstrate that the meta data used to build the dimensions of the experiment can be retrieved from the experiment context and used for reporting:



The resulting output shows for each scenario the meta-data that defines that scenario:



Recall that as the experiment executes, it utilizes multiple threads to execute the individual scenarios. Thus every experiment context consumer must be threadsafe. We have accomplished this by making each such consumer stateless. In practice, it is often necessary for experiment context consumers to be stateful and this can involve careful consideration of the use of synchronization and other concurrency issues. Fortunately, GCM provides a reporting plugin that deals with these issues and provides a general method for producing tabular reports.

# Chapter: Reports Plugin

The reports plugin implements an experiment context consumer that records output into tab-delimited text files via the java.nio library using blocking file writes. Three new concepts form the core of the reports:

1. Report Id – a unique identifier for each report
2. Report Header – the header content for the report
3. Report Item – the data content for each line in the report

The report id is a unique identifier used to mark every report item that the plugin processes and helps associate each item to the specific file where it will be recorded. Report items are a flexible list of values that have an associated report id as well as a report header used to build the header of the file.

# Chapter: Stochastics Plugin

asdf

# Chapter: Global Properties Plugin

asdf

# Chapter: People Plugin

asdf

# Chapter: Person Properties Plugin

asdf

# Chapter: Groups Plugin

asdf

# Chapter: Regions Plugin

asdf

# Chapter: Resources Plugin

asdf

# Chapter: Materials Plugin

asdf

# Chapter: Partitions Plugin

asdf