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

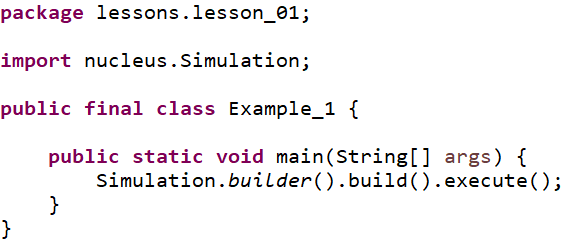
GCM also provides an experiment management system. Each plugin contains zero to many data objects that define the initial state of its actors and data managers. Each such data object may be altered freely. The complete set of all combinations (scenarios) of the variant plugin data objects form an experiment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

# Chapter 2: Getting Started

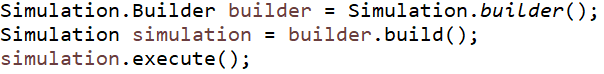
We start with a set of practical lessons that will help clarify the core concepts of GCM. The lessons generally build on one another and should be taken in order. You are encouraged to code along with the lessons.

## 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.



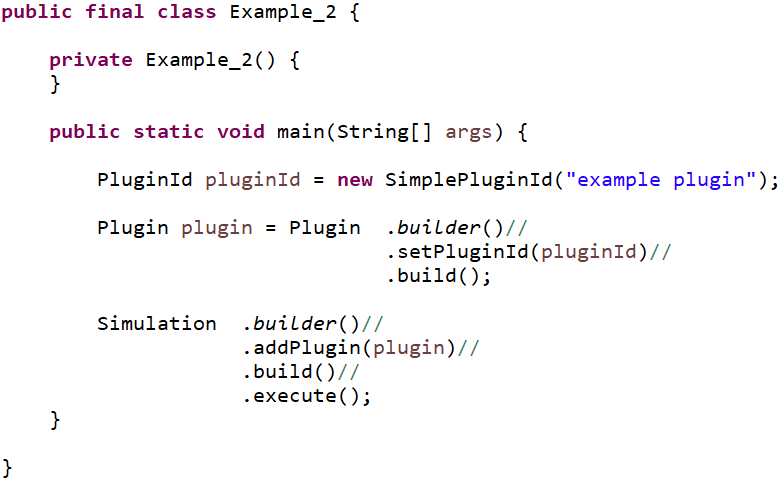
With this one line we have created and executed a simulation. Since the simulation had no actors or data managers there was nothing to do and so it terminated immediately. Let’s analyze the line in a more draw out form:



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

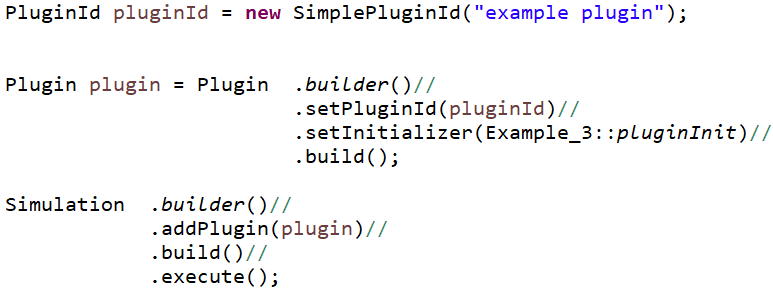
In all that follows, we will encounter various context objects. Contexts are interfaces into the simulation that are tailored to the thing using the context. For example, an ActorContext provides everything that an actor will need to interact with the simulation. Similarly, a DataManager context provides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

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

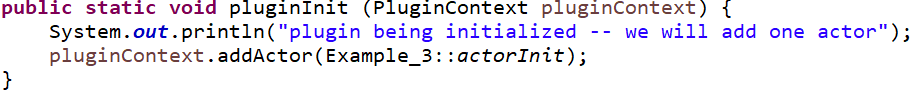
1. Add an actor to the simulation
2. Add a data manager to the simulation
3. Get plugin data

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.

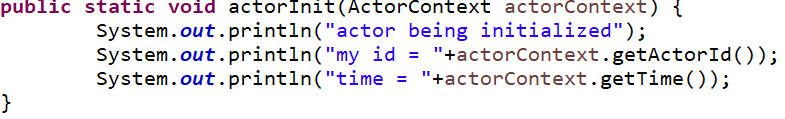
The next context will be the ActorContext. It provides actors with a wide array of capabilities that we demonstrate later. For now, the important takeaway is that being granted a context implicitly identifies the recipient as having a particular role in the simulation.



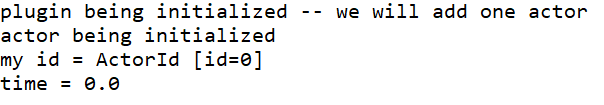
We are setting the plugin’s initializer. The initializer is a method that consumes a PluginContext and returns void. For this example, we use a static local method for our initializer:



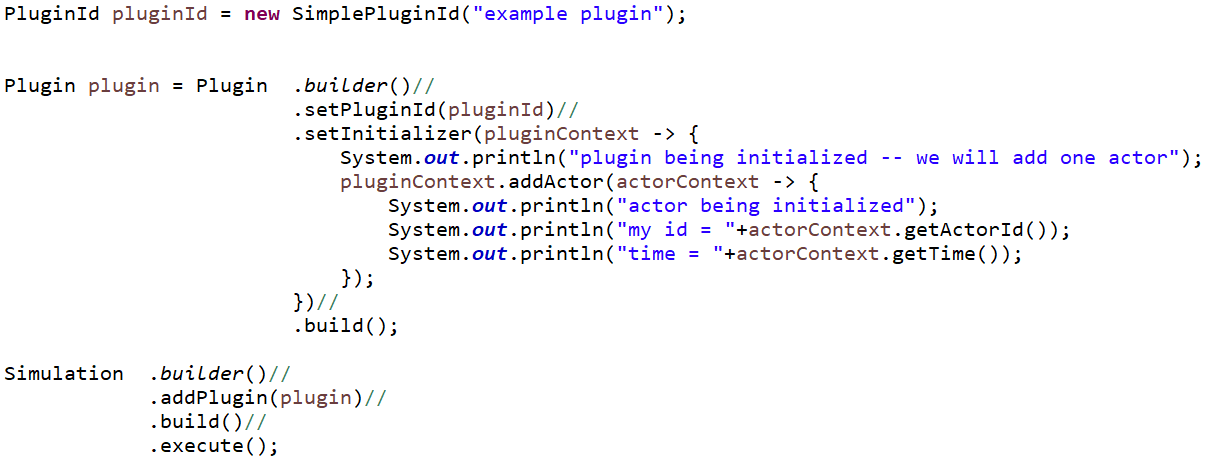
When the simulation starts up its execution, one of the first things it will do is to execute each plugin’s initializer to give the plugin an opportunity to add actors and data managers to the simulation before time and events begin to flow. Adding an actor is done with another consumer, but this time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

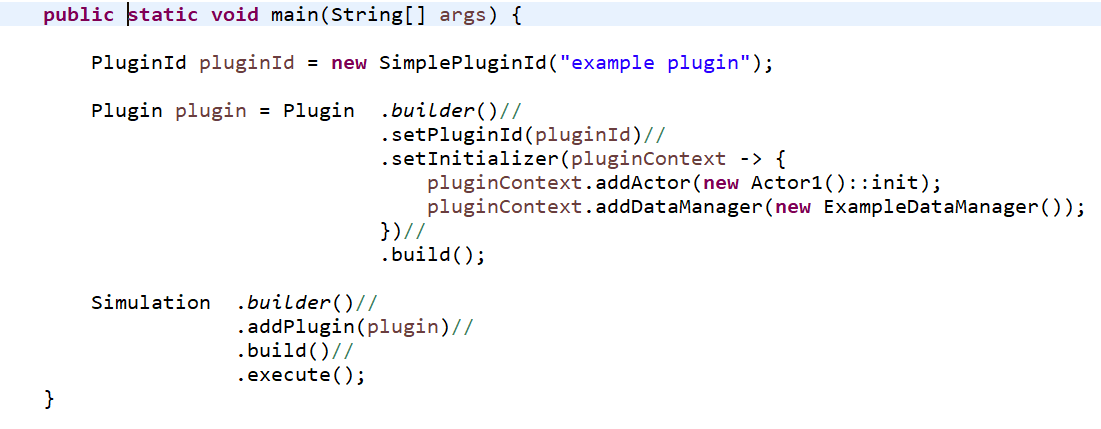
We extend the previous lesson by slightly altering the actor and adding a data manager. But first let’s list some of the attributes of data managers and actors to better understand the roles they play in the simulation.

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.

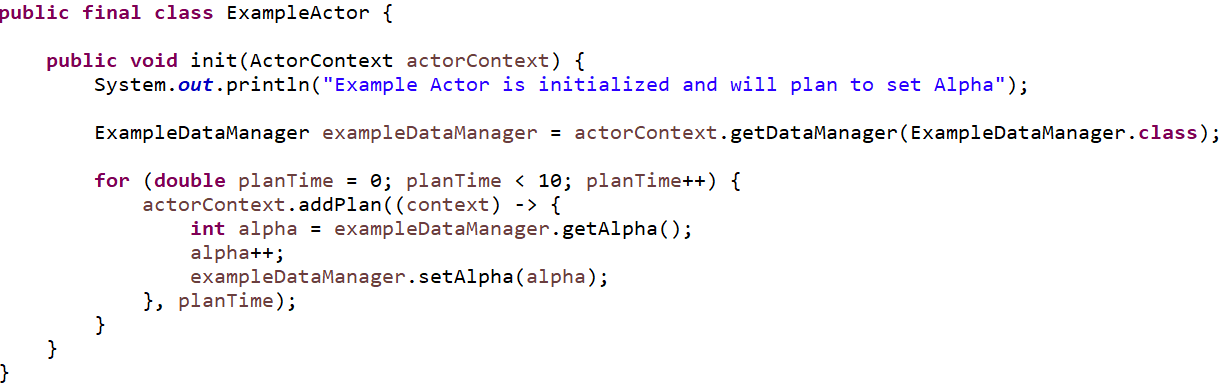
### Plans

In GCM, an actor can do three things:

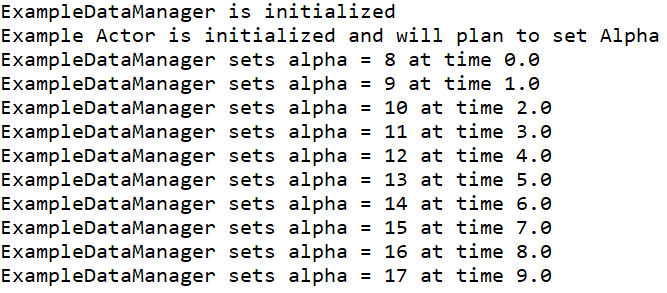
1. **Observe**: Observation can be done directly by gaining access to a data manager and then getting a value from that data manager. Observation can be done indirectly by subscribing to events. We will cover that option later.
2. **Act**: A mutation to some data manager’s managed data.
3. **Plan**: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

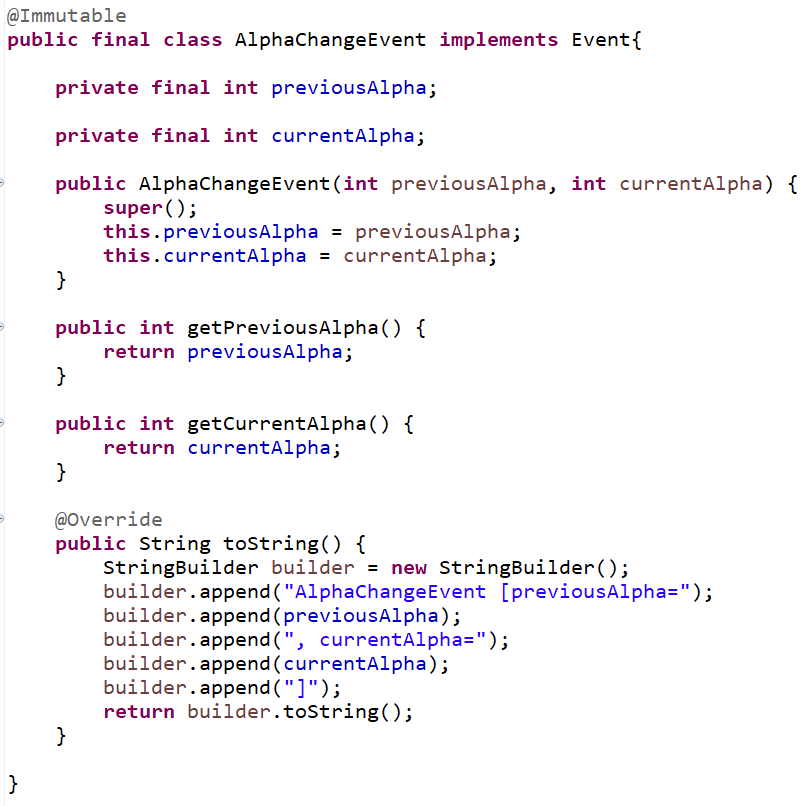


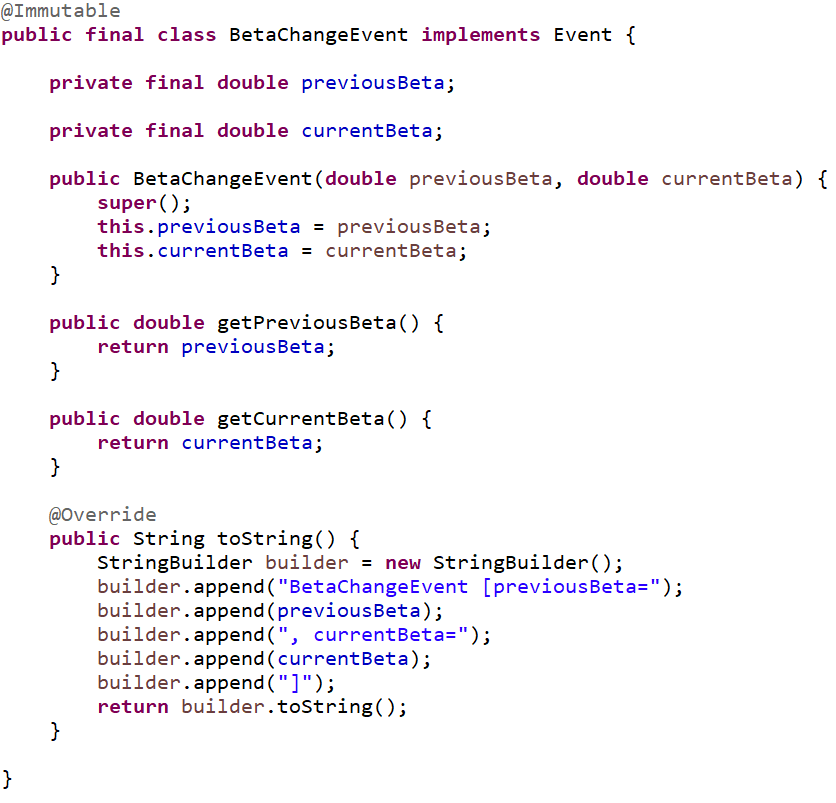
The output from the simulation is:



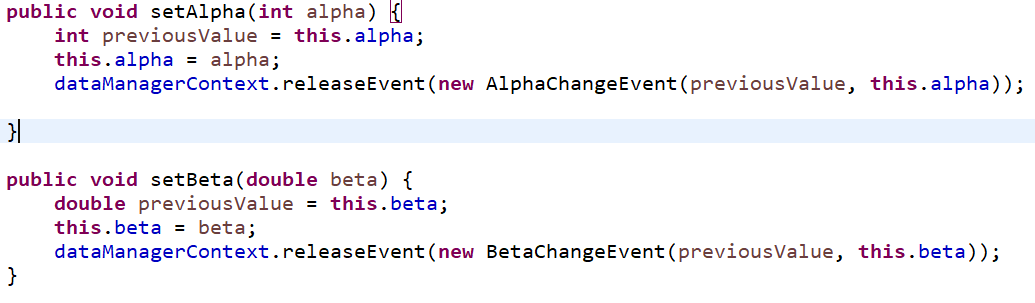
## Lesson 5: Introducing Events

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.



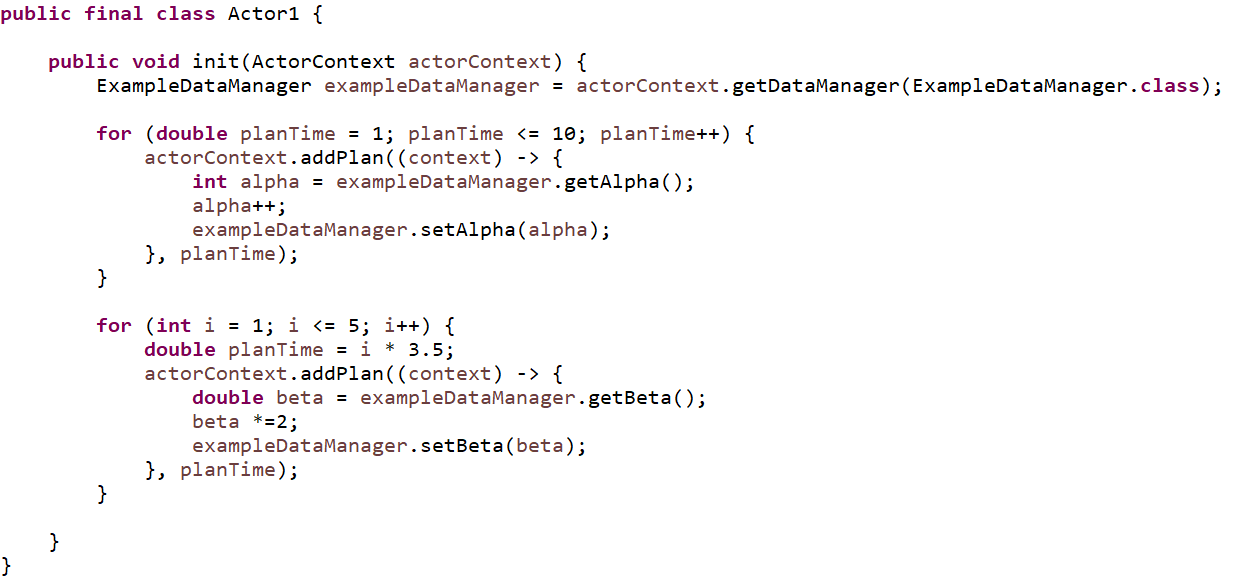


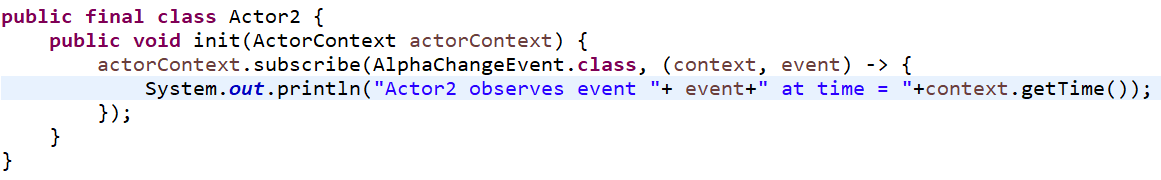
Each is generated by the ExampleDataManager when the alpha or beta values are mutated by releasing the events through the DataManagerContext to the simulation:

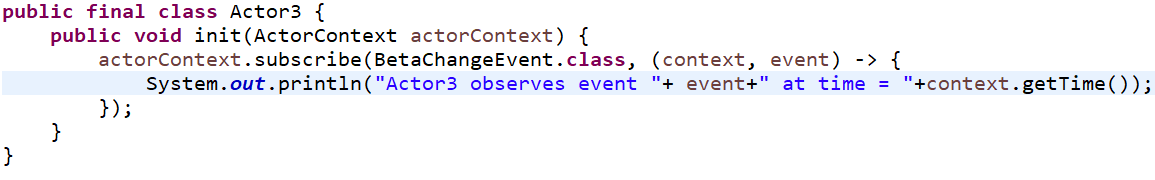


There are three actors in this example:

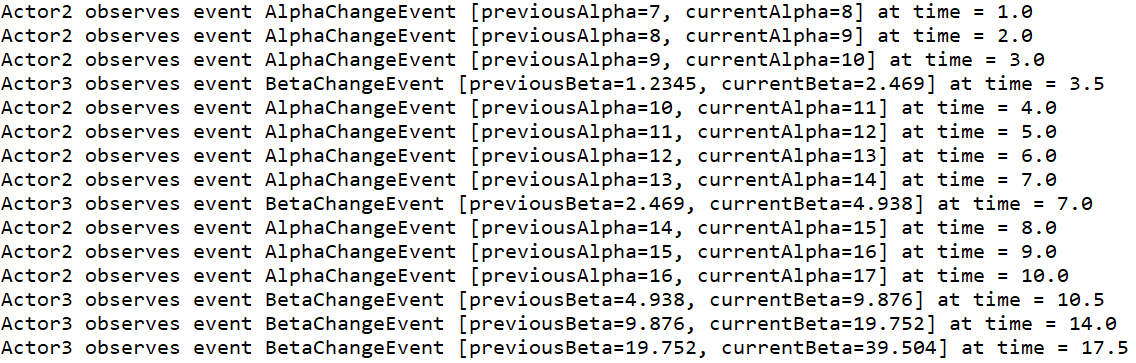
1. Actor1 makes changes to both the alpha and beta values at 1 and 3.5 day intervals respectively
2. Actor2 subscribes to AlphaChangeEvent events and reports to console what it receives
3. Actor3 does the same for BetaChangeEvent events







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



## Lesson 6: Introducing plugin dependencies

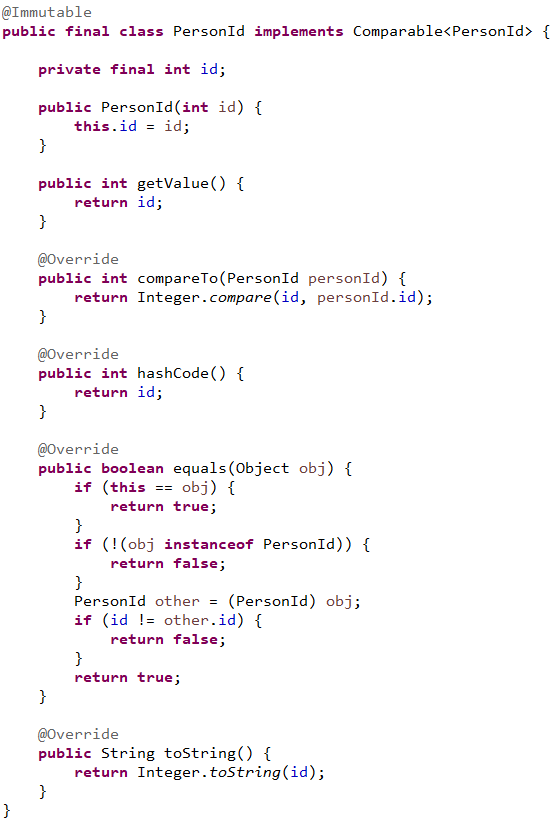
So far we have covered what actors and data managers do and that they are introduced into the simulation via plugins. Over the next lessons we take a closer look at the plugins. This lesson starts with creating a more realistic set of plugins each arranged into separate java packages.

* 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

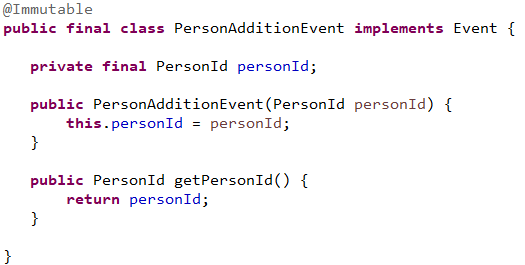
Here are the classes that implement this example:

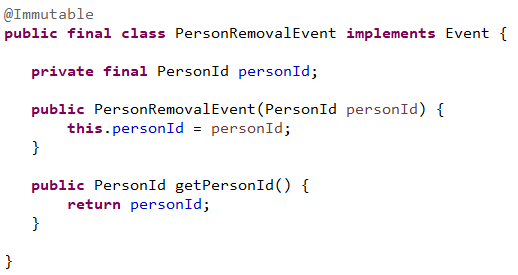
### People Plugin:

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.



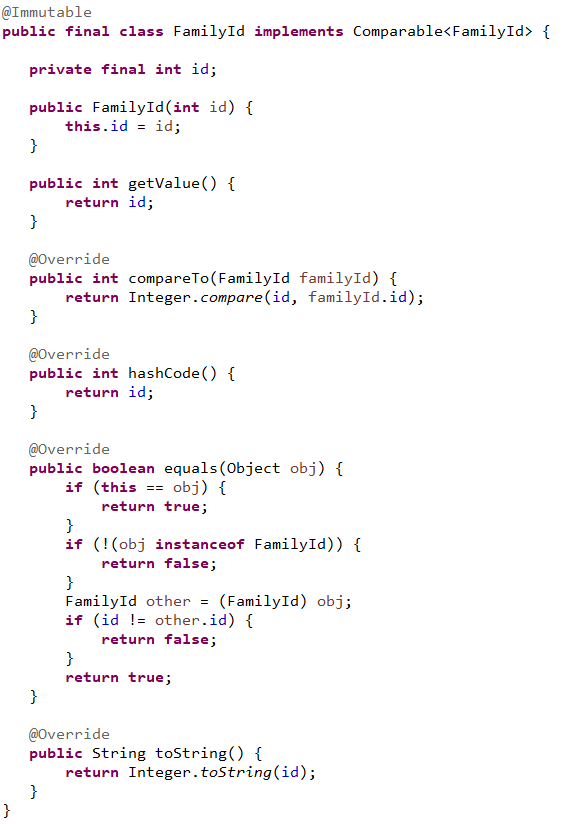


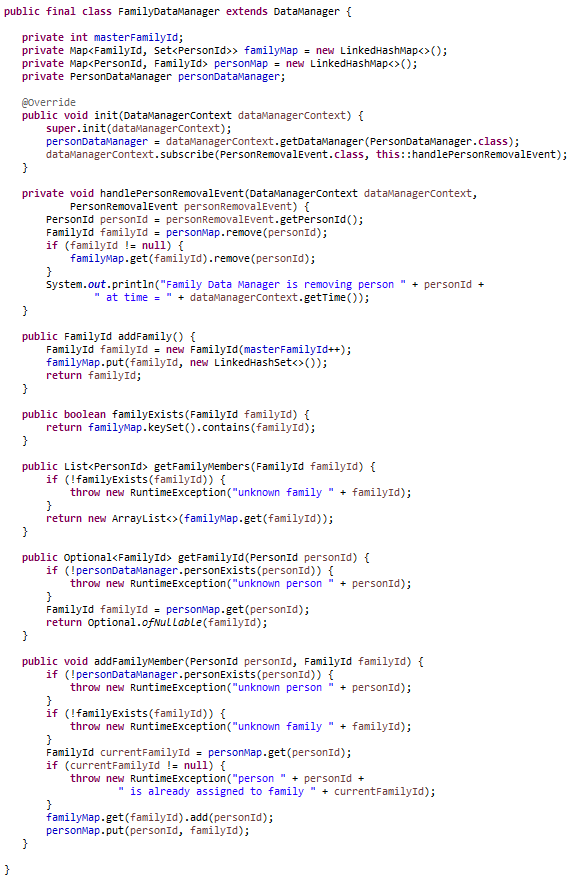




### Family Plugin

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

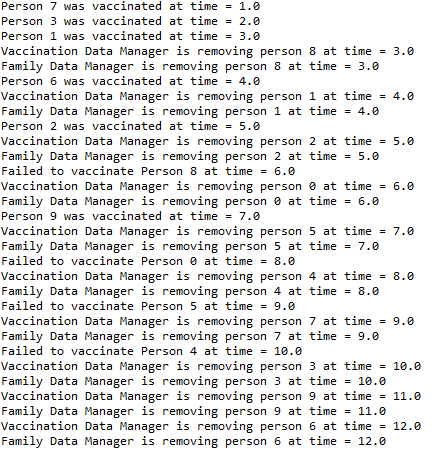
### Connecting the Plugins

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.

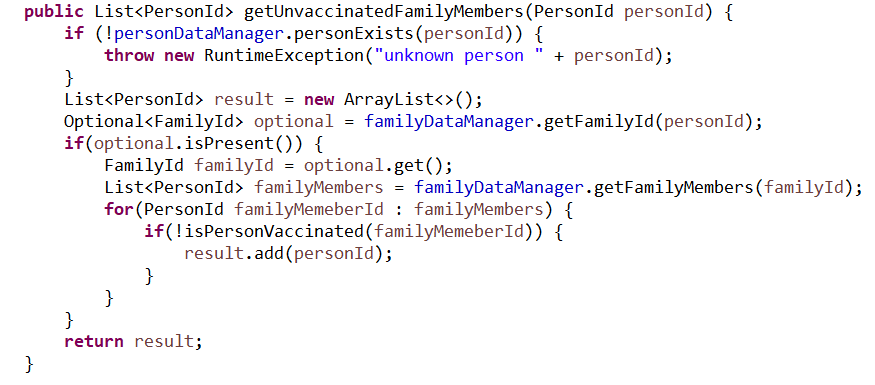
The resulting output:



## 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

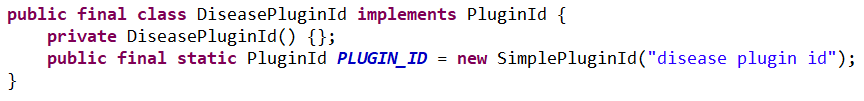
All plugin dependencies in GCM form similar directed, acyclic graphs (DAGs). There can be no loops in the dependency graph, but the graph does not have to be fully connected. The dependencies reflect the requirements of the data managers within a plugin to access data managers in other plugins. This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

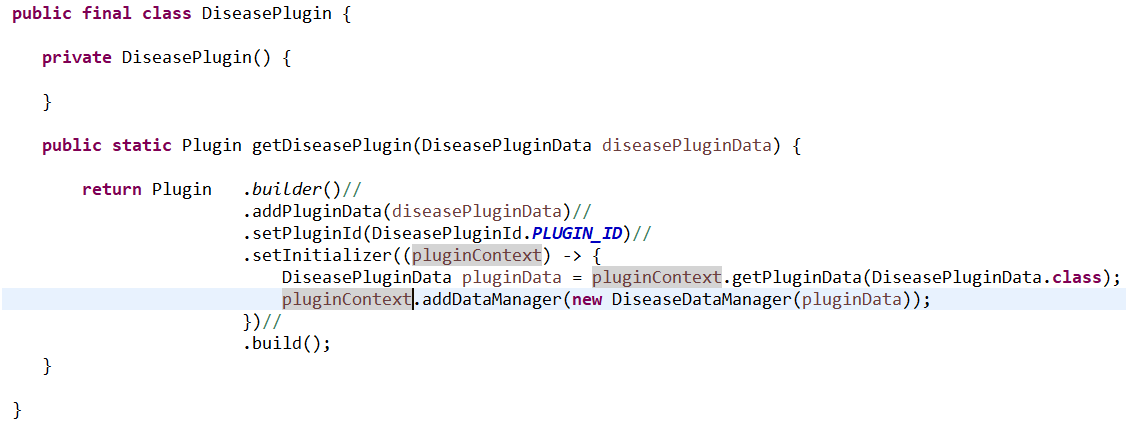
## Lesson 8: Plugin Packages

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.



You may have noticed that the initializer code above acquires the DiseasePluginData via the context rather than the instance passed to the getDiseasePlugin() method.

## Lesson 9: Introducing Experiments

Asdf

## Lesson 10: Experiment Output

# Chapter 3: Planning

asdf

# Chapter 4: Event subscription

asdf

# Chapter 4: Experiments

Asdf

# Chapter 5: GCM Life Cycle

Asdf

# Plugin: Reports

asdf

# Plugin: Stochastics

asdf

# Plugin: Global Properties

asdf

# Plugin: People

asdf

# Plugin: Person Properties

asdf

# Plugin: Groups

asdf

# Plugin: Regions

asdf

# Plugin: Resources

asdf

# Plugin: Materials

asdf

# Plugin: Partitions

asdf