JaamSim

Programming Manual

Revision 0.5
January 20, 2016

Table of Contents

1 Introduction

JaamSim (Java Animation Modelling and Simulation) is a discrete-event simulation software package first developed in 2002 as the foundation for simulation applications. JaamSim includes a drag-and-drop graphical user interface, 3D animation, and a full set of built-in objects for model building. It is object oriented, extremely fast, and scalable to the largest of applications. Windows, Linux, and OSX are all supported.

JaamSim represents the lessons learned from simulation projects we have performed around the world for more than 35 years. Further background information on JaamSim can be found in the JaamSim blog: www.jaamsim.com/blog.

JaamSim is free open source software, licensed under Apache 2.0. The latest version of the software and manuals can be downloaded from the JaamSim website: www.jaamsim.com. The source code is published on GitHub: www.github.com/jaamsim/jaamsim. Presentations and tutorials for JaamSim can be found at: www.youtube.com/user/javasimulation.

This Programming Manual proves detailed instructions on how to create new palettes of high-level objects that can be added on to JaamSim.

The user interface and basic objects provided with JaamSim are documented in a separate User Manual document. These features are common to all simulation models created with this software.

2 Getting Started

To develop new simulation components for JaamSim you will need to download the JaamSim source code and set up your computer to program in Java. The Java Development Kit and the Eclipse programming environment are the main tools that you will need for Java programming.

The instructions in this section assume that you are using a PC with the Windows operating system. Users of other systems, such Linux and OSX, can modify these instruction appropriately.

2.1 System Requirements

JaamSim will run on most modern computers that support OpenGL graphics version 2.1 or later. This includes laptop computers with Intel Core i5 and i7 series processors[1] with integrated graphics. NVIDIA graphics cards are preferred for best performance. AMD graphics cards are also supported.

The following system specifications are recommended for optimal performance with complex models:

- Intel Core i7 processor
- 16 GB of RAM
- NVIDIA GeForce or Quadro graphics card with at least 1 GB of RAM

2.2 JaamSim Source Code

The source code for JaamSim is available on GitHub.

1. Go to: https://github.com/jaamsim/jaamsim. Click on the "ZIP" button to download a zipped directory containing all the files.
2. Unzip the files and copy the "JaamSim-master" directory to your C: drive.

2.3 Java Runtime Environment and Java Development Kit

The Java Runtime Environment (JRE) is needed to execute your Java program. The JRE is sometimes referred to as the Java Virtual Machine. Although many computers with a Windows operating system come with the JRE already installed, it is best to uninstall this version and re-install the latest version.

1. The Java Development Kit (JDK) is needed to program in Java. The JRE and JDK are installed at the same time.
2. Download the JDK from www.oracle.com/technetwork/java/javase/downloads/index.html. Select the latest version of Java and download either the 32-bit or 64-bit installation file. For 64-bit windows users, the 64 bit version is the best choice.

Install the JDK by launching the downloaded installation file. Accept the option to install the JRE at the same time.

[1] Second generation or later Core i5 and i7 processors with integrated graphics are suitable for use with JaamSim. The integrated graphics capabilities for first generation Core i5 and i7 and earlier versions are not sufficient for JaamSim. A separate graphics card must be used with these processors.

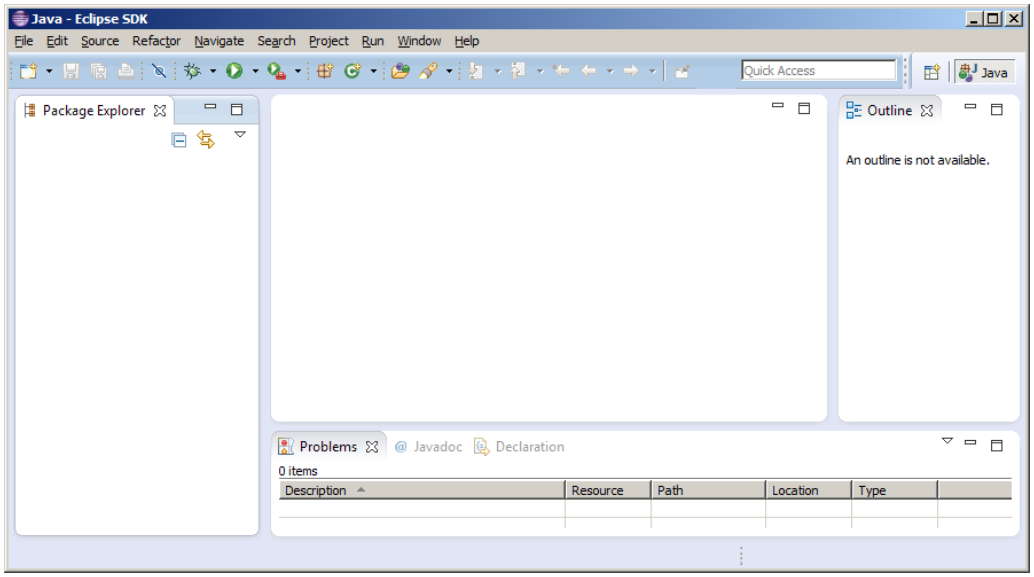
2.4 Eclipse

Eclipse is the recommended software development environment for Java. Its installation can be a bit daunting for the non-professional programmer, but it is well worth the effort in increased productivity once you start programming. We have described the installation and configuration process in more detail than is usual to make the installation as straightforward as possible.

2.4.1 Installation

1. Download Eclipse from www.eclipse.org/downloads/. The recommended version is "Eclipse Classic" (version 4.2.1 at the time of writing). For 64-bit windows users, the 64 bit version is the best choice.
2. Install Eclipse by unzipping the contents of the download to your C: drive.
3. Append the path for the JDK bin folder (e.g. C:\Program Files\Java\jdk1.7.0_45\bin) to the PATH environment variable. This must be the only entry to a Java bin folder. The following steps will take you to the environment variables in Windows 7:
    - click on the Start button
    - right click on the entry labelled "Computer" and select "Properties"
    - select "Advanced System Settings"
    - select the "Advanced" tab and click on the "Environment Variables" button
    - select "Path" and click the Edit button
    - append a semi-colon (";") followed by the path to the JDK bin folder
4. Confirm that Eclipse starts successfully by double clicking on eclipse.exe (in the C:\eclipse folder). An error message will appear if Eclipse cannot find your JDK files. If this happens check that the PATH was set up correctly in Step 3 above and that there is only one version each of the JDK and JRE installed on your computer.
5. On startup, Eclipse will prompt you for a workspace. Select the default. After launching successfully, Eclipse will appear similar to Figure 2-1.
6. Create a shortcut icon for eclipse.exe and place it on your desktop.

Figure 2-1 Eclipse Prior to Set-Up

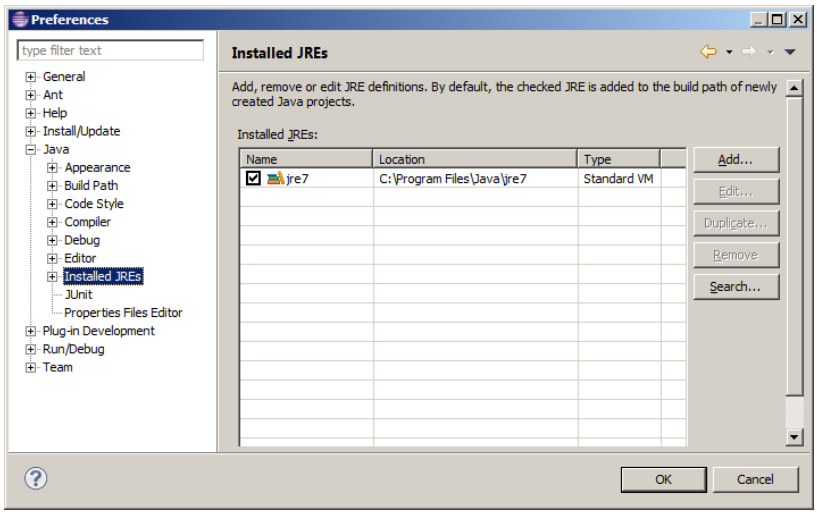


2.4.2 Set-Up the Java Runtime Environment

A number of additional procedures are required to set up Eclipse to work with the source code for JaamSim and the JAR files for supporting software.

1. Launch Eclipse by double clicking on eclipse.exe or on its shortcut icon.
2. Configure Eclipse for the Java installation:
    - o Select: Window -> Preferences -> Java -> Installed JREs
    - o Confirm that there is an entry for the Java Runtime Environment that you installed on your computer. If necessary, add any missing entries using the Add button. The Preferences tool should appear similar to **Figure 2-2** when you are finished.

Figure 2-2 Configuring Eclipse - Selecting the JRE



2.4.3 Set-Up the JaamSim Project

The next step is to set up Eclipse to work with the source code for JaamSim and the JAR files for supporting software.

Start a new Java project for the JaamSim source code:

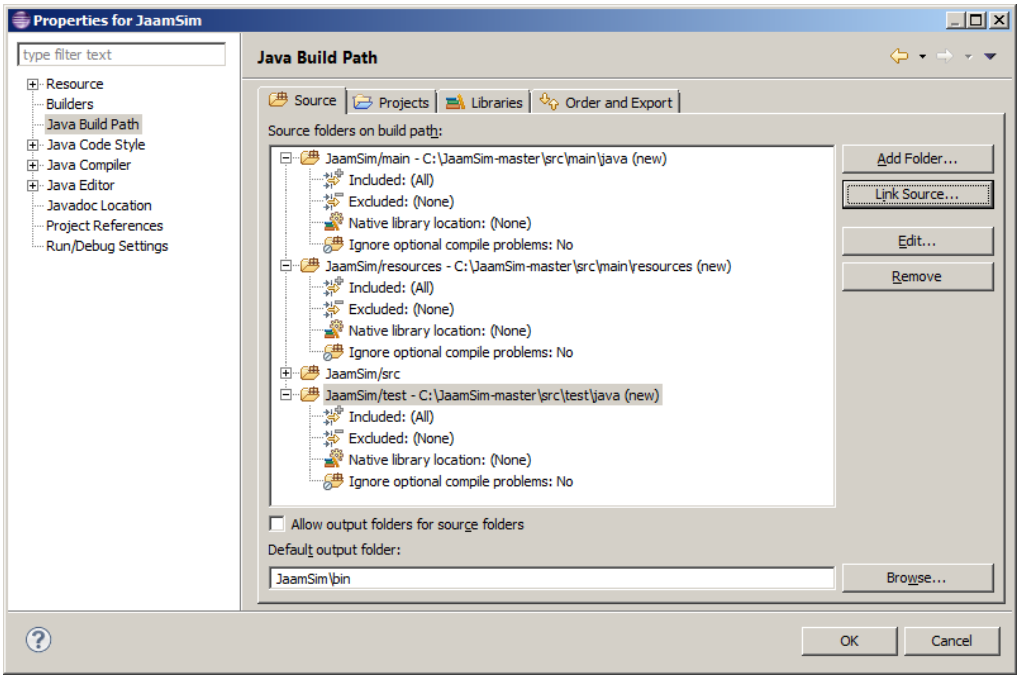
- o Select: File -> New -> Java Project
- o Name the new project JaamSim
- o Check “Use default location” for the project
- o Check “Use default JRE”
- o Click on the “Finish” button.

Link Eclipse to the JaamSim source code:

- o Select: Project -> Properties -> Java Build Path -> Source
- o In the panel labelled “Source folders on the build path:”, select the entry “JaamSim/src”, and click the “Remove” button.
- o Click on “Link Source” and enter the following information:
    - Linked folder location: C:\JaamSim-master\src\main\java
    - Folder name: main

- Click on the Finish button
- Click on “Link Source” and enter the following information:
  - Linked folder location: C:\JaamSim-master\src\main\resources
  - Folder name: resources
- Click on the Finish button
- Click on “Link Source” and enter the following information:
  - Linked folder location: C:\JaamSim-master\src\test\java
  - Folder name: test
- Click on the Finish button
- When the above steps are completed the dialog box should appear similar to Figure 2-3. Click on the OK button to close the window and complete the process.

Figure 2-3 Configuring Eclipse - Linking the Source Files

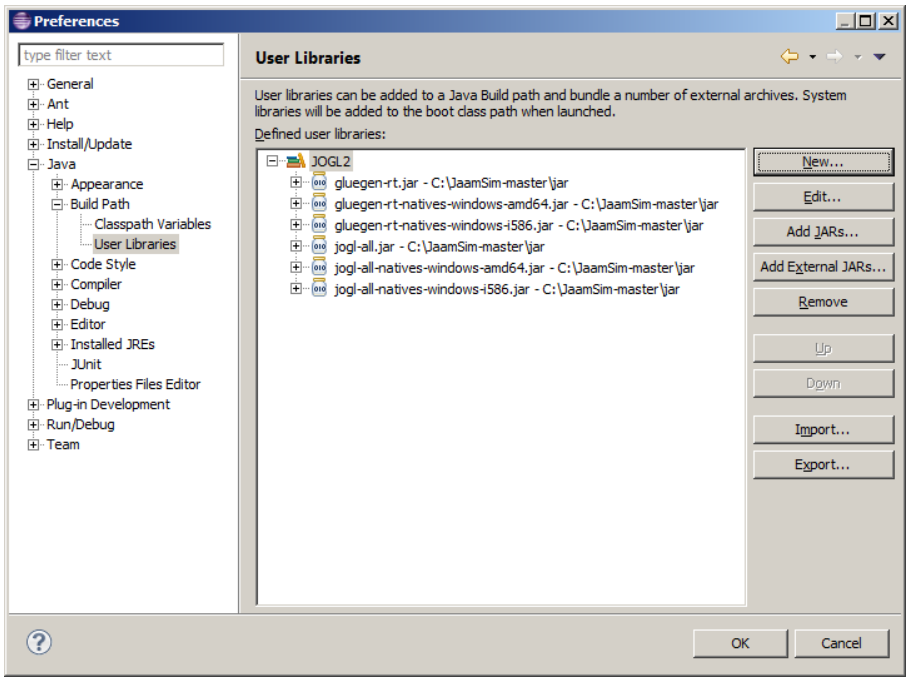


Set up the user library for JaamSim’s 3D rendering system (JOGL2)

- Select: Window -> Preferences -> Java -> Build Path -> User Libraries
- Click on “New” and name it JOGL2
- Click on “Add External JARs”
- Click on “Browse”, navigate to C:\JaamSim-master\jar, and select the following files:
  - gluegen-rt.jar
  - gluegen-rt-natives-windows-amd64.jar
  - gluegen-rt-natives-windows-i586.jar
  - jogl-all.jar
  - jogl-all-natives-windows-amd64.jar
  - jogl-all-natives-windows-i586.jar
- Click on “Open” to add these JAR files. The User Libraries should now look similar to Figure 2-4.

- Click on “OK” to close the Preferences window.

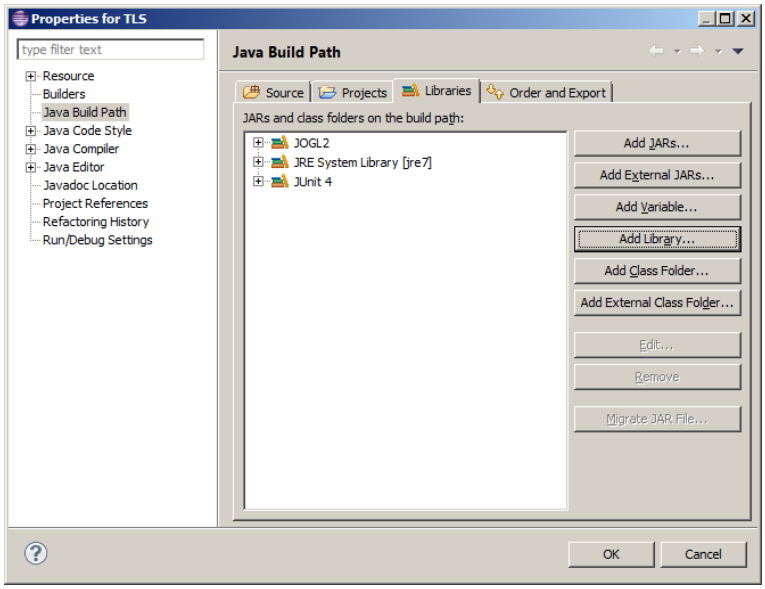
Figure 2-4 Configuring Eclipse - User Libraries



Add the new Libraries to the Build Path:

- Select: Project -> Properties -> Java Build Path -> Libraries
- Click on “Add Library” and select “User Library”
- Select JOGL2 from the list
- Click on “Finish” to complete the addition
- Click on “Add Library” and select “JUnit”
- Select JUnit4 from the list
- Click on “Finish” to complete the addition. The Java Build Path window should now look similar to Figure 2-5.
- Click on “OK” to close the Preferences window.

Figure 2-5 Configuring Eclipse – Adding Libraries



Select the "Main" method to execute when JaamSim is launched from Eclipse:

- Select: Run -> Run Configurations … -> Main
- Next to the "Main Class" heading, click on the Search button
- Select the entry for "GUIFrame"
- Click on the "Close" button to accept the change.

Test the installation by running JaamSim from Eclipse. Click on the green run button to launch the software. After a short delay to load the code, JaamSim should launch normally.

2.4.4 Recommended Optional Settings

Avoid trailing whitespace:

- Select: Window -> Preferences -> Java -> Editor -> Save Actions
- Select the checkbox for "Perform the selected action in save"
- Uncheck the boxes for "Format source code" and "Organize imports"
- Check the box for "Additional actions"
- Click the "Configure" button
- Under the "Code Organizing" tab, check the box for "Remove trailing whitespace"
- Click "OK"
- Now back at the "Save Actions" screen, click "Apply" and then "OK"

3 Simulation Objects and Methods

The ability to provide simulated time and to coordinate multiple simultaneous processes is the distinguishing feature of a simulation modeling language. JaamSim provides a number of convenient tools for handling simulated time that are fully-integrated with the underlying Java programming language. JaamSim provides the programmer with all the tools required to write highly readable and easy-to-modify code.

3.1 Overview

The basic object classes used to implement discrete-event logic within Java are given in **Table** 3-1.

Table 3-1 Basic Simulation Objects

Object	Description
Entity	The basic object for simulation. Can be permanent or temporary during the simulation run and can be active or passive in the model logic.
Process	A sub-class of Thread that allows an entity to execute a series of methods in simulated time while other entities execute their own series of methods.

Note that we have separated the two objects Entity and Process. When an Entity is playing an active role in the simulation, it will have one or more Processes underway. When it is playing an inactive role or is temporarily dormant, it will have no Process underway.

Unlike most other simulation software, we have made the distinction between starting a new method, which is done in series with the original method, and starting a new process, which is done in parallel with the original method. Other simulation languages return control to the original method if the called method is halted by a wait – equivalent to starting a new process each time a method is called.

A new process can be created and started using the Process methods in **Table** 3-2.

Table 3-2 Starting a Process

Method	Description
startProcess(method, arg1, arg2, …)	Calls the given method and passes in the specified arguments, i.e. this.method(arg1, arg2, …). However, it differs from a simple method call in that a new process is created, allowing the method to be executed in parallel to the original method (in simulated time), rather than in series.
scheduleProcess(dur, method, arg1, arg2, …)	The same function as startProcess except that the given method is called after the specified delay. The new process is created at the end of the delay, to minimize the number of active threads.

Table 3-3 Scheduled and Conditional Waits

Method	Description
simWait(dur, pri)	Stops the execution of the current method for the given duration in seconds of simulated time. Can be placed anywhere within a method and can be used multiple time within a method.

While (condition) {
  waitUntil();
}
waitUntilEnded();
	Code pattern used to create a conditional wait. Two new methods, waitUntil() and waitUntilEnded() are used. Stops execution of the current method until the given condition is FALSE.

scheduleLast()	Stops execution of the current method until all other events scheduled for the present simulated time have been executed.

JaamSim allows both the process- and event-orientation to be used freely in the construction of a simulation model:

- simWait is the key method for writing a process-orientated simulation model.
- scheduleProcess is the key method for writing an event-oriented simulation model.

An event-oriented model is more efficient than a process-oriented model because it minimises the number of active processes and avoids context switching. However, it is easier to follow complex model logic in a process-oriented model and in many such models, it is the model's methods themselves that limit execution speed rather than the overhead of managing threads. The best approach is to use an event-oriented style (the scheduleProcess method) whenever possible and save the process-oriented style (the simWait method) for the more complex parts of the model. Often, it is useful to prototype new objects using simWait methods and then, after it works correctly, optimize selected portions by converting the code to use scheduleProcess.

Table 3-4 Interrupting and Terminating a Process

Method	Description
getProcess()	Returns the active process executing the current method. The methods interruptProcess and killProcess are the only ones that require a process to be identified.
interruptProcess(processName)	Interrupts the given process and causes its next event to be executed immediately.
killProcess(processName)	Interrupts the given process and terminates it.

3.2 Simulation Time

The current simulated time in seconds can be obtained within the model by using the Process method getSimTime() which returns a double value, e.g.

```
double t = Process.getSimTime();
```

The unit for simulation time is seconds.

Internally, simulation time is maintained as an integer value in order to avoid the accumulation of round-off error during a simulation run. This internal long-valued time is related to the double-valued

simulation time by GraphicSimulation input keyword SimulationTimeScale (see the JaamSim User Manual). The default value for SimulationTimeScale is 1000 so that one unit of internal time is equal to one millisecond.

3.3 Starting a New Process

3.3.1 Execution Order

JaamSim allows a method to be executed either in series or in parallel in simulated time to the calling method. In standard Java, a called method is executed in series with the method that called it. For example, consider the following code for methods aaa() and bbb().

```
int n;

public void aaa() {
   n = 1;
   this.bbb()
   n = 4;
}

public void bbb() {
   n = 2;
   Process.simWait(1.0);
   n = 3;
}
```

For this case, the simWait method in bbb() causes time to advance by one hour before control is passed back to method aaa(). If method aaa() is executed at time = 0, then the property n takes on the following sequence of values:

- time = 0: n = 1
- time = 0: n = 2
- time = 1: n = 3
- time = 1: n = 4

When a method is called using a startProcess command, it is executed in parallel to the method that called it. The modified code for methods aaa() and bbb() illustrates this behaviour.

```
int n;

public void aaa() {
   n = 1;
   Process.startProcess("bbb");
   n = 4;
}

public void bbb() {
   n = 2;
   Process.simWait(1.0);
   n = 3;
}
```

In this case, a new process is started for the purpose of executing method bbb(), and method aaa() continues its execution at time 0 while method bbb() is waiting for one hour. Now, the sequence of values for property n is:

- time = 0, n = 1
- time = 0, n = 2
- time = 1, n = 4
- time = 1, n = 3

3.3.2 Passing Arguments

In the above examples, the called method bbb() had no arguments. For methods that do take arguments, they can be appended after the method name in the startProcess call. For example, the code

```
Process.startProcess("bbb", arg1, arg2, … , argN);
```

would create a new process and execute the following code:

```
this.bbb( arg1, arg2, … , argN);
```

In this example, the arguments arg1 … argN can be any type of object required by method bbb.

3.3.3 Scheduling a Process to Start in the Future

This method is being rewritten at present.

```
Process.scheduleProcess(dur, "bbb", arg1, arg2, …, argN);
```

3.4 Scheduled Waits

Simulated time can be advanced in a simulation model using a number of different statements. The simplest type of delay is one whose duration is known in advance. The Process method for a specified duration delay is:

```
Process.simWait(duration);
```

Where duration is a double value denoting the duration of the delay in simulated time. Execution of the method is resumed at the end of this time.

JaamSim delays execution of the method at this point by suspending the current process and creating an event which stores the simulated time at which the process is to be resumed. The EventManager maintains a stack of future events sorted in order of scheduled execution time. Events scheduled themselves at the same time will be evaluated in stack order (First In - Last Out), that is, the last event scheduled for a given time will be the first one to be executed.

It is also possible to influence the execution sequence of events scheduled at the same time by specifying an event priority using the statement:

```
Process.simWait(duration, priority);
```

Where duration is the double-valued wait duration and priority is an integer priority. The default priority of an event is 5. Events scheduled at the same time are executed in order of increasing

priority value, e.g. priority 1 events are executed before priority 2 events. Priority must be restricted to the range 1 – 10. Priorities 11 and 12 are used for special purposes.

3.5 Conditional Waits

A conditional wait is of unknown duration, but instead is determined by logical condition that must be satisfied in order to end the wait. The syntax for a conditional wait is:

```
while (booleanExpression) {
    Process.waitUntil();
}
Process.waitUntilEnded();
```

Where booleanExpression can be any expression that evaluates to a boolean value. The boolean expression must be false to end the wait.

The conditional wait expressions are evaluated at the end of each simulation time. If multiple events have been scheduled at the same time, all of these events will be executed before the conditional expressions are checked, and possibly executed.

Conditional waits are evaluated in first-in-first-out order based upon the order they were added to the conditional queue. Once a condition is satisfied, the program continues execution from that spot until completion, or another wait is reached in the code. EventManager then continues checking the rest of the queue and does not re-evaluate the already checked conditionals until the next time advance.

The statements for a conditional wait work in the following manner. The method waitUntil() suspends the active process and places it in the conditional queue maintained by eventManager. At each time at which any events have been scheduled, after executing the scheduled events, EventManager restarts each process one-by-one. On restarting, the process hits the right brace of the while expression and re-checks the booleanExpression. If the expression is true, the thread is re-suspended. If it is false, the thread passes out of the while statement and executes waitUnitEnded(), which removes the process from the conditional queue.

Note that booleanExpression is checked before the thread is suspended. If it is false on first evaluation, then the thread is never halted at all. It is also possible to rewrite the statements as follows so that the condition is not checked until the thread has been suspended and all other events at that time have been executed and all other conditionals have been checked:

```
do {
    Process.waitUntil();
} while (booleanExpression)
Process.waitUntilEnded();
```

Similarly, if a thread is to be delayed so that it is the last to be executed at a given time, the booleanExpression can be omitted altogether:

```
Process.waitUntil();
Process.waitUntilEnded();
```

3.6 Schedule Last

A scheduleLast wait does not increase simulation time at all. It waits for all other events to be evaluated before restarting. Such events are scheduled with a priority of 11 or 12. Regular events should only be scheduled with a priority of 1-10 so the regular scheduler will place them below the

regular events on the stack at the current time. These events can then be ordered in LIFO (top of other last events, with priority 11) of FIFO (below other last events, with priority 12). These functions are called as with other scheduler functions scheduleLastLIFO() and scheduleLastFIFO().

3.7 Interrupting or Terminating a Future Event

It is sometimes necessary in a simulation model to interrupt or terminate a future event that has already been scheduled using Process.simWait(). A future event that is interrupted is executed immediately, while one that is terminated is simply discarded.

To accomplish either of these actions, the programmer must keep a reference to the process corresponding to the event to be terminated. For example, the following code saves a reference to the current process just before it is suspended:

```
processholder = this.getProcess();
Process.simWait(duration);
processholder = null;
```

Where processholder is the variable maintaining the reference to the process to be terminated. Note that this reference must be establish before the thread has be suspended by the simWait().

The future event can be interrupted by the statements:

```
Process.interruptProcess(processholder);
processholder = null;
```

It can be terminated by the statements:

```
Process.terminateProcess(processholder);
processholder = null;
```

In all of the above code segments, processholder is set to null immediately after the reference is no longer needed. This is done to prevent a dangling reference to the process which would prevent proper cleanup once the process has finished execution. It is very important that the programmer not leave dangling references to processes. For efficiency, processes managed by EventManager are pooled for re-use later in the simulation. This leaves the possibility of a process being terminated even though it has completed execution in one place and has been re-started somewhere else in the program. If this were to occur, it would be a very difficult to debug error.

3.8 Discrete-Event Logic

JaamSim uses the same basic discrete-event logic as other simulation packages. Two master lists are maintained by the software:

- **Future Events**. A list of future events sorted into order of increasing event time. Events with the same event time are sorted in order of increasing priority variable (priority 1 events are executed before priority 2, and so on). If both the event time and priority are the same, events are sorted in reverse order in which they were added. That is the last event added is the one that is executed first.
- **Conditional Events**. A list of conditional events sorted in the order in which they were added. That is, the first conditional event that was added is the one that is tested first.

Events are executed in the following manner:

1. Set the current time to zero.
2. Events whose event time is equal to the current time are pulled from the Future Event list and executed one by one. Each event may add additional future events, so the Future Event list will often change with each event.
3. Each conditional event on the Conditional Event list is tested one by one. If the condition is satisfied, the event is removed from the list and is executed.
4. Advance the current time to that for the next event on the Future Event list and, if this time is less than the end of run time, return to (2).
5. End of the simulation

4 3D Graphics

JaamSim features a new 3D rendering system that is programmed in Java and is integrated with the rest of the JaamSim code to provide fully-interactive, high-performance graphics. The renderer is entirely shader based and can make best use of the latest games-type graphics cards. It is not necessary to use the more expensive "workstation" type graphics cards normally required for engineering software[2].

More to come

Table 4-1 Graphics Methods for DisplayEntity

Method	Description
setPosition(pos)	Sets the position to the given x, y, z coordinates (Vec3d)
setSize(size)	Sets the size to the given x, y, z dimensions (Vec3d)
setAlignment(align)	Sets the alignment to the given x, y, z values (0.0 – 1.0) (Vec3d)
setOrientation(euler)	Sets the orientation Euler angles to the given values (Vec3d)

Table 4-2 Renderer Interface Methods for DisplayEntity

Method	Description
updateGraphics(simTime)	Sets the position, size, alignment, and orientation (as required) based on the object's logical state at the given simulation time. To achieve smooth motion, the method must update the graphics appropriately for any given simulation time, not just event times. Should be called only by the Renderer.

[2] Many of the standard engineering applications use the old-style fixed graphics pipeline instead of shaders, which severely limits their performance on games type graphics cards. Workstation cards and their drivers are specially designed to compensate for this shortcoming.

5 Model Inputs

For simulation models with hundreds of objects and thousands of inputs, it is essential to provide a good method for specifying model inputs. Furthermore, the model input files must be readable enough for them to be audited readily.

JaamSim provides a powerful system for handling the inputs to a simulation model. The system allows simple models to be constructed quickly and can be scaled to handle arbitrarily complex models.

- Keyword definitions are shown interactively as tool tips in the Input Editor.
- Drop down menus in the Input Editor for most inputs.
- Unit conversions are done automatically when the input is read
- Input files for configuring a model are human readable and can be edited directly by the user
- Input files can be created and edited interactively by drag and drop and the Input Editor without losing any comments and special formatting added directly by the user.

New objects programmed by a user can take advantage of all these features with very little effort:

- New keywords are automatically added to the Input Editor and provided with drop-down menus as appropriate.
- Documentation is entered in the code as an annotation next to each keyword definition. The keyword definition appears automatically in the user interface as a tool tip when the user mouses over the keyword in the Input Editor.

5.1 Input Objects

Input objects were created to standardize the various types of inputs required by our simulation models.

Input Object	Type of Inputs
BooleanInput	A single Boolean input value, e.g. { FALSE }
BooleanListInput	A sequence of Boolean input values, e.g. { FALSE TRUE }

Input Object	Type of Input
ValueInput	A single double input value with or without a unit, e.g. { 1.0 m } or { 1.0 }
ValueListInput	A list of double valued inputs with or without a unit, e.g. { 1.0 2.0 m } or { 1.0 2.0 }

Input Object	Type of Input
SampleInput	A single number or an object that returns a number, such as a probability distribution, a time series, or a calculation object, e.g. { 1.0 m } or { Distribution1 } or { TimeSeries1 } or { Calculation1 }.
SampleListInput	A list of numbers or a list of objects that return a number, e.g. { 1.0 2.0 m } or { Calculation-1 Calculation-2 }.

Input Object	Type of Input
IntegerInput	A single, dimensionless, integer input value, e.g. { 1 }.
IntegerListInput	A list of dimensionless, integer inputs values, e.g. { 1 2 }.

Input Object	Type of Input
ColorInput	A single RGB or color name input value, e.g. { 256 0 0 } or { lavender }.
ColorListInput	A list of RGB or color name inputs values, e.g. { { 256 0 0 } { red } }.

Input Object	Type of input
EntityInput	A single Entity input, e.g. { Queue1 }.
EntityListInput	A list of Entity inputs, e.g. { Queue1 Queue2 }
EntityListListInput	A table of Entity inputs, e.g. { { Queue1 Queue2 } { Queue3 Queue4 } }

Input Object	Type of Input
Stringinput	A single String input, e.g. { ‘Example String’ }.
StringListInput	A list of String inputs, e.g. { ‘First String’ ‘Second String’ }.
StringChoiceInput	Chooses a String from a list of valid Strings.

Input Object	Type of Input
KeyInput	A single combination of an Entity and a double, e.g. { Queue1 2.0 }.
KeyListInput	A list of Entity and double combinations, e.g. { { Queue1 5.0 } { Queue2 1.0 } }

5.2 Program Structure for Inputs

Input objects are defined for an object class and then added to each instance. The following example shows the standard structure we used:

```
public class NewObject extends DisplayEntity {

    // DEFINE THE INPUT OBJECTS
    @Keyword(description = "Description of the first keyword",
             example = "NewObject-1 Key1 { Queue-1 }")
    private final EntityInput<Queue> key1;

    // define any other inputs here.

    // DEFINE THE PROPERTIES
    // define the properties here.

    // ADD THE INPUTS TO EACH INSTANCE THAT IS CREATED
    {
        key1 = new EntityInput<Queue>(Queue.class, "Key1", "Cat", null);
        this.addInput(key1, true);

        “Add the other inputs here.
    }

    // CONSTRUCTOR
    public NewObject() {

    }
}
```

In this example, the keyword “Key1” is added to the object “NewObject”. This keyword expects an input of a Queue object (a sub-class of Entity). The input object is defined by the statement:

```
private final EntityInput<Queue> key1;
```

This defines a property named “key1” that is an EntityInput of type Queue. The definition and example that will appear in the Input Editor tooltip is given by the @Keyword notation.

Next, an instance of EntityInput is created and assigned to the key1 property by the statement:

```
key1 = new EntityInput<Queue>(Queue.class, "Key1", "Cat", null);
```

The constructor for the EntityInput specified the type of Entity expected (Queue.class), the name assigned to the keyword (“Key1”), the input category for the keyword (“Cat”), and the default value (null).

For other types of InputObject, it is not necessary to specify an object class in the constructor since the InputObject is already specialized to one object type. The general for the constructor is:

```
key1 = new xxxxInput("Key1", "Cat", def);
```

where the strings “Key1” and “Cat” are the keyword name and category, respectively, and def is the default value for the input.

In the case of IntegerInput and ValueInput, it is possible to specify the valid range for the input using the setValidRange method:

```
key1.setValidRange(min, max);
```

where min and max are the minimum and maximum values for the numerical input. If the setValidRange method is not use, min and max default to negative and positive infinity, respectfully.

For numerical inputs, the type of unit (time, distance, etc.) must be specified using the setUnitType method:

```
key1.setUnitType(ut.class );
```

where the ut is any valid unit type that is defined for JaamSim. Units are discussed in more detail in the next section.

The present value of an input object is returned by the method getValue(). To avoid any trouble with initialization, it is best to use the input object and the getValue() method directly in the code each time the input is required. There is no need to assign a local variable or another property just to hold the input value.

5.3 Unit Conversion

Numerical inputs must specify their unit type (time, distance, etc.) using the setUnitType method:

```
key1.setUnitType(ut.class);
```

where the ut is any valid unit type that is defined for JaamSim. For example, the code

```
@Keyword(description = "The length of the object.",
   example = “Object-1 Length { 10.0 m }")
   private final DoubleInput length;
{
length = new DoubleInput(“Length”, “Basic Inputs”, 0.0);
length.setValidRange(0.0, Double.POSITIVE_INFINITY);
length.setUnitType(DistanceUnit.class);
this.addInput(length, true);
}
```

would create an input keyword “Length” and identify it as expecting an input with the units of distance. In this case, a valid input would be:

```
Object  Length { 5.0 m }
```

An input without any units would generate an input error message, e.g.

```
Object  Length { 5.0 }
```

would be an invalid input for this keyword.

The following unit types have been defined at present. Additional unit types can be created by adding new sub-classes of Unit.

Unit Type	Internal Unit
AccelerationUnit.class	m/s2
AngleUnit.class	rad
AngularSpeedUnit.class	rad/s
AreaUnit.class	m2
CostRateUnit.class	$/s
CostUnit.class	$
DensityUnit.class	kg/m3
DimensionlessRateUnit.class	/s
DimensionlessUnit.class	
DistanceUnit.class	m
EnergyDensityUnit.class	J/m3
EnergyUnit.class	J
LinearDensityUnit.class	kg/m
MassFlowUnit.class	kg/s
MassUnit.class	kg
PowerUnit.class	J/s
PressureUnit.class	Pa
SpecificEnergyUnit.class	J/kg
SpeedUnit.class	m/s
TimeUnit.class	s
ViscosityUnit.class	Pa-s
VolumeFlowUnit.class	m3/s
VolumeUnit.class	m3

Note that a dimensionless numerical input should be assigned the unit type DimensionlessUnit.class.

The internal units given in the above table are the ones used for internal calculations. Wherever possible, we have used SI units for JaamSim's internal calculations.

The individual units within a unit type are defined by creating instances of the unit type. For example, the units of metres and kilometres are defined by creating the instances "m" and "km" using the following inputs:

```
Define  DistanceUnit  { m  km }
m    ConversionFactorToSI { 1        }
km   ConversionFactorToSI { 1000     }
```

Many units are created in this way by the autoload.cfg file, which is loaded automatically on startup. The autoload file is discussed in more detail in Section 7.1.

Inputs entered by the user in another unit are converted automatically to the internal unit using the value for the ConversionFactorToSI. For example, an input of { 1.5 km } would be converted to 1500 m when the input was first read by the program.

5.4 Using an Input

The value for an input can be obtained by using the Input method getValue(), which will return the appropriate object type for the keyword. For example, if the property testKey is used to store the Input object corresponding to the keyword TestKey, then the present value for this input can be obtained by executing the following code:

```
double x = testKey.getValue();
```

For this example, TestKey was assumed to contain a double, but the same getValue() method can be used to obtain the present value for any type of Input.

It is best to use the getValue() method every time an input is needed instead of storing a cached value. This approach ensures that the current value for the input is always used and, in many cases, allows an input to be changed in the middle of a simulation run without restarting.

5.5 Input Checking and Model Startup

Basic checking of a model input to ensure that its format is valid and that its value is within a valid range is done when the input is first read from the input file or when a new value is entered in the Input Editor. Further checking is done in the validate() method for each object when the model is actually started, i.e. when the Start button is clicked. This validate() method is intended to ensure that there are no missing inputs that would cause errors during execution and that the various inputs are consistent with one another. Any new objects created by the user will need to include a suitable validate() method.

For example, the following is a simple validate method for the EntityGenerator class:

```
@Override
public void validate() {
    super.validate();

    // Confirm that the next entity in the chain has been specified
    if (nextComponent.getValue() == null) {
        throw new InputErrorException( "The keyword NextEntity must be set." );
    }
}
```

InputErrorException is a subclass of ErrorException that generates an error message box in the user interface. The message is also added to the log file for the run. In batch mode, the error message is logged without generating the message box and the run is terminated.

Other methods that need to be provided for new objects are:

- earlyInit() – Initialisation of the object that is required on start up or after restarting. All initialization required for a simulation run should be performed in this method. Initialization should not be done in the object's constructor since it will not be repeated on restart.
- startup() – Actual starting of any active objects.

For each method, it is important to invoke the super-class' method prior to any further processing. For example, the earlyInit() method should be similar to the following:

```
@Override
public void earlyInit() {
    super.earlyInit();

    //Initialization specific to this object
}
```

If no specific code is required for the object then the method should be left out altogether so that it defaults to its super-classes method.

6 Model Outputs

Model outputs are coded using an @Output annotation, which is similar to the @Keyword annotation used for inputs.

Outputs differ from inputs in that they change continuously during the simulation run. Rather than storing each output as a property and ensuring that it is updated prior to use, we have chosen to identity the outputs with methods that calculate and return the desired value. This approach ensures that the output value is up to date and avoids any unnecessary calculation of outputs that are infrequently needed.

The following example shows the basic pattern for outputs:

```
@Output(name = "AverageServiceTime",
 description = "The average time per entity serviced to date.",
    unitType = TimeUnit.class)
public double getAverageServiceTime(double simTime) {
   return totalWorkingTime/numberProcessed;
}
```

In this example, AverageServiceTime is name of the output that will appear in the Output Viewer. The description field will appear as tooltip pop-up when the user mouses over this entry in the Output Viewer. The unit field indicates that this output has units of time.

The method getAverageServiceTime calculates the value of this output. All outputs must take a time argument for reasons that will be discussed below. Numerical outputs should always be returned as a double.

Outputs should always be freshly calculated from the object's internal state. Storing and using a pre-calculated output introduces unnecessary computation and is an invitation to logical errors caused by out of date outputs.

Outputs take a time argument so that, when appropriate, the output can be calculated at any time, not just at the event times. This approach allows the output to vary continuously between events, which is helpful for outputs that are used in the 3D display. For example, if the process of travelling between one location and another is modeled by a start-travelling event and an end-travelling event, then to show smooth motion, it is necessary to calculate the object's position at any intermediate time. This can be done only if the renderer passes the present time to the output method.

The following example show a situation where the time argument can be used.

```
@Output(name = "FractionCompleted",
 description = "The fraction of the service time that has been completed.",
    unitType = DimensionlessUnit.class)
public double getFractionCompleted(double simTime) {
if( busy ) {
          return (simTime - startOfServiceTime) / serviceTimeInput.getValue();
   }
   else {
          return 0.0;
   }
}
```

In this example, the internal property "startOfServiceTime" was updated when the Server started processing its present customer and its state "busy" was set to true. By using the time argument simTime in the calculation, the output FractionCompleted can increase smoothly from 0 to 1 as the

customer is serviced. This output could be used by a bar gauge or some 3D action to show the service being performed.

7 Graphical User Interface

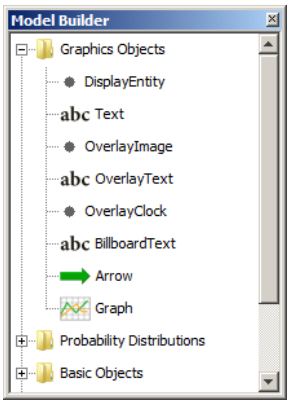
The previous sections have described how to create new simulation objects within JaamSim. The task for this section is to integrate these new objects within the JaamSim user interface so that they can be used in the same way as the built-in objects.

An automatically-loaded input file is used to configure many of the built-in components of JaamSim. The file in question, autoload.cfg, can be found in the folder

```
C:\JaamSim-master\src\main\resources\resources\inputs\autoload.cfg
```

Along with various other tasks, the autoload.cfg file sets up the Palettes and connects the class files to the objects in the drag-and-drop tools. A Palette is a set of drag-and-droppable objects that appear in the Model Builder. In the following figure, “Graphical Objects”, “Probability Distributions”, “Basic Objects”, etc. are Palettes.

Figure 7-1 Palettes in the Model Builder



The following excerpt from graphic.inc, one of the file loaded by autoload.cfg, shows how the Palette is created and how the classes DisplayEntity, Arrow, and Text are set up:

graphic.inc (exerpt):

```
Define Palette { 'Graphics Objects' }
Define ObjectType { DisplayEntity  Arrow  Text }

DisplayEntity  JavaClass { com.jaamsim.Graphics.DisplayEntity }
Arrow          JavaClass { com.jaamsim.Graphics.Arrow }
Text           JavaClass { com.jaamsim.Graphics.Text }

DisplayEntity  Palette { 'Graphics Objects' }
Arrow          Palette { 'Graphics Objects' }
Text           Palette { 'Graphics Objects' }
```

The first two lines define the “Graphic Objects” palette and the three object types: DisplayEntity, Arrow, and Text. Next, the JavaClass keyword is used to connect these object types with the Java classes in the program and Palette keyword is used to assign each object to the Graphics Objects palette.

A user can add new objects to JaamSim by creating a similar file and appending the appropriate “Include” statement to the end of autoload.cfg. The following example shows the input files for a hypothetical new palette “New Objects” containing the object type “MyComponent”.

Autoload.cfg file:

```
Include sim.inc
Include units.inc
Include observers.inc
Include graphics.inc
Include displayModels.inc
Include basicObjects.inc
Include newObjects.inc         ”New line added by the User”
```

NewObjects.inc file:

```
Define Palette { 'New Objects' }
Define ObjectType { MyComponent }

MyComponent  JavaClass { com.user.MyComponent }
MyComponent  Palette { 'New Objects' }
```

7.1 Adding New Default Objects

The autoload.cfg file is also used for creating all the default objects in JaamSim. In all but a few cases, instances of high-level objects are created in the autoload.cfg or default.cfg files, not in the code. By restricting the code to define object class and not the instances, it becomes much easier to code the logic for stopping and restarting a model as well as to clear one model and to load another.

Default objects can be either built into JaamSim and will always appear, or they can be optional and saved along with the user's other inputs. Built-in objects are defined in the autoload.cfg file, while optional object are defined in the default.cfg file. The user is free to add additional objects to either file as required.

7.1.1 Built-In Objects

Default objects that will appear in every model should be defined in the autoload.cfg file. For example, all the built-in units in JaamSim are defined in the units.inc file (one of the files included in autoload.cfg). The following excerpt from this file shows the inputs used to define all the time units:

Units.inc file (exerpt):

```
Define TimeUnit       { s min h d w y ms }

s   ConversionFactorToSI { 1      }  Description { ‘Seconds’ }
min ConversionFactorToSI { 60     }  Description { ‘Minutes’ }
h   ConversionFactorToSI { 3600   }  Description { ‘Hours’ }
d   ConversionFactorToSI { 86400  }  Description { ‘Days’ }
w   ConversionFactorToSI { 604800 }  Description { ‘Weeks’ }
y   ConversionFactorToSI { 31536000 }  Description { ‘Years’ }
ms  ConversionFactorToSI { 1.0e-3   }  Description { 'Milliseconds' }
```

Additional units or other objects can be added to this file by the user.

7.1.2 Optional Objects

Default objects that appear when JaamSim is first loaded, but will be cleared when an input file is loaded, should be defined in the default.cfg file:

```
C:\JaamSim-master\src\main\resources\resources\inputs\default.cfg
```

These objects can be included when building a new model or they can be deleted in the usual way if they are not wanted. Any of these objects that are retained will be saved in the input file along with the user-defined objects.

The version of default.cfg included with JaamSim defines a number of objects:

- coordinate axes (XYZ-Axis)
- coordinate grid (XY-Grid)
- default view window (View1)
- model title overlay (Title)
- simulation time and date overlay (Clock)

The default.cfg file that creates these objects is shown below:

Default.cfg file:

```
"This file is loaded when no configuration file has been specified.

Simulation Description { 'Simulation run control inputs' }

" Start the model in Real Time mode
Simulation RealTime { TRUE }

" Set the Pause Time to the default
Simulation PauseTime { }

" Select the tools to show on startup
Simulation ShowModelBuilder   { TRUE }
Simulation ShowObjectSelector { TRUE }
Simulation ShowInputEditor    { TRUE }
Simulation ShowOutputViewer   { TRUE }
Simulation ShowPropertyViewer { FALSE }
Simulation ShowLogViewer      { FALSE }

" Create grid
Define ColladaModel { Grid100x100 }
Grid100x100 ColladaFile { <res>/shapes/grid100x100.dae }

Define DisplayEntity { XY-Grid }
XY-Grid Description { 'Grid for the X-Y plane (100 m x 100 m)' }
XY-Grid DisplayModel { Grid100x100 }  Size { 100 100 m }  Movable { FALSE }

" Create axis
Define ColladaModel { Axis }
Axis ColladaFile { <res>/shapes/axis_text.dae }

Define DisplayEntity { XYZ-Axis }
XYZ-Axis Description { 'Unit vectors' }
XYZ-Axis Alignment { -0.4393409 -0.4410096 -0.4394292 }  Size { 1.125000  1.1568242
1.1266404 m }  Movable { FALSE }
XYZ-Axis DisplayModel { Axis }

Define View { View1 }
View1 Description { 'Default view window' }
View1 ShowWindow  { TRUE }
View1 SkyboxImage { '<res>/images/sky_map_2048x1024.jpg' }

" Create Clock and Title overlays
```

```
Define TextModel { TitleTextModel ClockTextModel }
TitleTextModel Description { 'Text style for the Title' }  FontColour { 150 23 46 }
FontStyle { BOLD   }
ClockTextModel Description { 'Text style for the Clock' }  FontColour { 51 51 51  }
FontStyle { ITALIC }

Define OverlayText { Title }
Title Description { 'Title for the simulation model' }
Title Format { 'Model Title' }
Title TextHeight { 18 }  ScreenPosition { 15 15 }
Title DisplayModel { TitleTextModel }

Define OverlayClock { Clock }
Clock Description { 'Simulation date and time (no leap years or leap seconds)' }
Clock DateFormat { 'yyyy-MMM-dd HH:mm:ss.SSS' }  StartingYear { 2014 }
Clock AlignBottom { TRUE }  TextHeight { 10 }  ScreenPosition { 15 15 }
Clock DisplayModel { ClockTextModel }
```

Additional objects can be added by the user to this file.

8 Program Structure

Key components of JaamSim:

Table 8-1 Key Object Classes

Class	Description
GUIFrame	The Control Panel user interface that appears on the computer screen.
Simulation	Controls the starting and execution of the simulation run. For programming convenience, it is a sub-class of Entity.
EventManager	Controls the discrete-event logic and time keeping. An instance of EventManager is created by Simulation and stored as the static property "root" of Entity.
Process	A sub-class of Thread that allows an Entity to execute a series of methods in simulated time while other entities execute their own series of methods.
Entity	The basic simulation object that has access to the discrete event logic.
DisplayEntity	A sub-class of Entity that adds 3D graphics.

Table 8-2 GUIFrame Methods

Method	Description
main()	The main program that is executed on the launch of JaamSim or of an application of JaamSim such as TLS. Reads the configuration file and calls the static Simulation method start().
startSimulation()	Called when the Resume/Pause button on the Control Panel is clicked to start or resume the simulation run.
pauseSimulation()	Called when the Resume/Pause button on the Control Panel is clicked to pause the simulation run.

Table 8-3 Simulation Methods

Method	Description
start()	Starts the simulation run. Calls validate() for each Entity and then schedules the first event.
startModel()	Calls startUp() for each Entity.
doEndAt(t)	Schedules the end of the simulation run at time t and waits for this time to arrive. When the the end time is reached, the method doEnd() is called for each Entity. If the input ExitAtStop is set, then JaamSim is terminated.

In JaamSim, the discrete-event logic is executed by the EventManager object. Its properties eventStack and conditionalList are used to maintain the future event and conditional event lists, respectively.

Table 8-4 EventManager Methods

Method	Description
run()	Executes the discrete event logic using the Future Events list (eventStack). Calls evaluateConditionals() to test the conditional events.
evaluateConditionals()	Tests the conditional events using the Conditional Event list (conditionalList).

Table 8-5 Entity Methods

Method	Description
validate()	Checks that the Enity's inputs are consistent with each other and with those for any other Entities. Any checking that does not involve another keyword or Entity is done when the input was first read from the input file or was entered in the Input Editor. The validate method is overwritten for each subclass of Entity that requires its inputs to be validated. The default method for Entity does nothing.
earlyInit()	Initialises the Entity based on its inputs, but does not start any processing yet. The earlyInit() method is overwritten for each subclass of Entity that requires initialisation. The default method for Entity does nothing.
lateInit()	Performs a second round of initialisation after all the entities in the model have executed their earlyInit() method.
startUp()	Starts any processes required for the Entity to begin the simulation run. Only active Entities require a startUp() method. The startUp() method is overwritten for each subclass of Entity that requires starting. The default method for Entity does nothing.
doEnd()	Executes any post-run preparation that is required prior to the output report being written.