403 Poyntz Ave., Suite B
Manhattan, KS 66502, USA
Phone: +1-785-770-8511
Email: support@thunderheadeng.com
Web: https://www.thunderheadeng.com

Pathfinder Scripting API Manual

Version: 2020-4
Last Modified: 2020-09-24

Table of Contents

Chapter 1. Getting Started

Pathfinder scripts can be edited within the Pathfinder user interface and are stored with other model data inside the save file (PTH). Once a script has been added to a model, it will automatically be used any time anyone runs that model.

Since this is an experimental feature, a flag must be set to enable the script editor. This flag can be set by either running Pathfinder from a console window (recommended when creating/debugging scripts) or by editing the Windows Start Menu shortcuts.

To enable the script editor, it is necessary to add the following VM argument when running Pathfinder:

```
-Dex_enable_scripting
```

The above form of the argument can be used as-in when part of a batch file (e.g. monte carlo runs), but requires a modification when running Pathfinder as part of a shortcut or from the command line:

```
-J-Dex_enable_scripting
```

The `-J` instructs the wrapper executable to unpack the argument as a VM parameter.

For example:

```
pathfinder.exe -J-Dex_enable_scripting
```

Once this flag has been enabled, an option will be added to the **Model** : **Model › Edit Custom Scripts** menu If the script editor is not enabled, the presence of custom scripts will trigger a warning when loading the PTH file.

The following sections describe how to use the API scripting feature. When working with scripts, please keep in mind the following details:

- Scripts run when the simulation is initialized, before any movement. To run code after initialization, you must add callback functions.
- Scripting is not compatible with simulation restart.
- All scripts share the same global scope, creating a situation that shares the same advantages and disadvantages with JavaScript development for web browsers. Namely, you can conveniently create library classes that are shared across multiple script blocks as defined in Pathfinder, but you can just as easily overwrite important variables by accident. *JavaScript namespacing* is the topic to learn more about if you are having trouble with the latter case.

Chapter 2. Introduction

The Pathfinder API provides low-level access to the internals of a Pathfinder simulation.

It consists of four modules:

Name	Object	Description
Simulation Control	`api.simctl.v1`	Simulation management and callbacks
Geometry	`api.geometry.v1`	Access simulation objects like doors and rooms
Agents	`api.agents.v1`	Access agent/occupant data during the simulation
Input/Outout	`api.io.v1`	Access the out/err/in streams for I/O.

The **v1** on the end of the object name gives access to a specific version of that API and will allow us to keep legacy scripts operational if we introduce API-breaking changes.

The following convenience variables will be used in all examples in this documentation.

```
var sim = api.simctl.v1;
var geom = api.geometry.v1;
var agents = api.agents.v1;
var io = api.io.v1;
```

Chapter 3. Core API

The Core API describes operations that are explicitly supported by Pathfinder's API.

3.1. Simulation Control

The Simulation Control API makes it possible to register callbacks to execute code during the simulation.

```
var sim = api.simctl.v1;
```

3.1.1. onUpdate(Function(t))

Register a callback function. It will be invoked once on each simulation timestep.

The function accepts one parameter:

Parameter	Type	Description
t	double	Current simulation time

```
sim.onUpdate( function (t) {

   // do this every time step

});
```

3.1.2. onExit(Function(t))

Register a callback function. It will be invoked once at the end of the simulation.

```
sim.onExit( function () {

   // do this once after the simulation has ended
   // recommended for closing PrintStream

});
```

3.2. Geometry

The Geometry API provides methods to access geometry objects like doors, rooms, and measurement regions. In addition, methods are provided to open and close doors.

The geometry methods use the objects of type `ANode` which is a class that includes both doors and rooms and type `Region` which represents a measurement region.

```
var geom = api.geometry.v1;
```

3.2.1. close(ANode)

Close a door.

```
geom.close( door );
```

3.2.2. find(String)

Find a single door or room by name.

```
var door = geom.find("Door to Atrium");
```

3.2.3. findAll(String)

Find all named doors or rooms that match a regular expression.

Additional information on regular expressions in Java see java regex.

```
var doors = geom.findAll("Door to Atrium");
```

3.2.4. open(ANode)

Open a door.

```
geom.open( door );
```

3.3. Agents

The Agents API provides methods to access agents during the simulation.

```
var agents = api.agents.v1;
```

3.3.1. find(String): OccAgent

Find a single agent by name.

```
var agent = agents.find("00028");
```

3.3.2. findAll(Region): List<OccAgent>

Find all agents whose *center point* is within a measurement region.

```
var regionA = geom.find("Region-A");
var agentsInRegionA = agents.findAll( regionA );
```

3.3.3. findAll(String): List<OccAgent>

Find all named agents that match a regular expression.

Additional information on regular expressions in Java see java regex.

```
var agentsNamedStaff = agents.findAll( "staff-??" );
```

3.3.4. getAllAgentsEver(): List<OccAgent>

Returns a list of all agents that have ever been in the simulation, including those that have exited.

```
var agents = agents.getAllAgentsEver();
```

3.4. Input/Output

`api.io.v1`

The Input/Output API provides methods to write to and read from the simulation input and output streams.

```
var io = api.io.v1;
```

3.4.1. openPrintStream(String): PrintStream

This method can be used to open a PrintStream to a file in the current simulation's working directory. Streams should be closed when the simulation exits to prevent file locking issues.

```
var fileout = openPrintStream("myfile.tsv");
```

3.4.2. out

Simulation standard output stream. Text will appear alongside other console output.

```
io.out.println("message");
```

3.4.3. err

Simulation standard error stream. Text will appear alongside other console output and may be colored red depending on the editor.

```
io.err.println("message");
```

3.4.4. in

Simulation standard input stream. Possibly a way to collect user input from the console or piping together applications...?

```
(TBD)
```

Chapter 4. Extended API

The Core API describes operations that are explicitly supported by Pathfinder's API, but it's possible to access a much wider range of objects. An example of a common situation is when dealing with the `findAll()` methods that return an instance of `java.util.List`. Another example is the `OccAgent` object that is returned by the `find()` method. In addition, you can create arbitrary instances of objects using the API.

In general:

- If you are trying to deal with objects that you receive as a result from a Core API function, there should be some guidance in that section of the Core API documentation.
- If you are interested in creating an instance of a class from scratch, the general pattern is to use the `Java.type()` method to look up a type definition and then use that type to create new instances.

Create a file using the Java platform library.

```
var PSType = Java.type("java.io.PrintStream");
var fileStream = new PSType("myfile.txt");

fileStream.println("Hello");

filestream.close();
```

Basically, you can write any program you want by using the extended API because it encompasses both JavaScript and the Java platform. The details of how really dive into these is beyond the scope of this document, but if you have specific questions or requests for code examples, please send us an email.

Chapter 5. Examples

The following examples combine elements of the script API.

5.1. Door Open/Close

This example finds a specific door by name. The door will operate on a 40 second cycle. For the first 30 seconds the door will be closed. Then open for 10 seconds. Then the cycle will repeat.

This example makes use of the modulus (%) operator which performs a division and gives the remainder rather than the quotient.

```
var sim = api.simctl.v1;
var geom = api.geometry.v1;

var door = geom.find("Station Entrance");

sim.onUpdate( function (t) {

    var tc = t % 40.0;
    if (tc < 30) {
        geom.close(door);
    }
    else
    {
        geom.open(door);
    }
});
```

5.2. Control Doors by Region Count

This example opens and closes doors based on the number of agents that are standing inside measurement regions.

```
var sim = api.simctl.v1;
var geom = api.geometry.v1;
var agents = api.agents.v1;

var rA = geom.find("Region A");
var doorToA = geom.find("Door to Zone A");

var rB = geom.find("Region B");
var doorToB = geom.find("Door to Crossroad");

sim.onUpdate( function (t) {

    var agentsInRegionA = agents.findAll(rA);
    if (agentsInRegionA.size() > 100)
    {
        geom.close(doorToA);
    }
    else
    {
        geom.open(doorToA);
    }

    var agentsInRegionB = agents.findAll(rB);
    if (agentsInRegionB.size() > 100)
    {
        geom.close(doorToB);
    }
    else
    {
        geom.open(doorToB);
    }
});
```

5.3. Print Interval Output

Print output to the console at 10 second intervals.

```
var sim = api.simctl.v1;
var geom = api.geometry.v1;
var io = api.io.v1;

var rA = geom.find("Region A");
var rB = geom.find("Region B");

var dtOutput = 10.0;
var nextOutput = 0.0;

// header
io.out.println("Time\tA\tB");

sim.onUpdate( function (t) {

  if (nextOutput <= t)
  {
    var nA = agents.findAll(rA).size();
    var nB = agents.findAll(rB).size();

    // data rows
    io.out.println(t +"\t" + nA +"\t" + nB);

    nextOutput += dtOutput;
  }

});
```