Get Involved

java-net Project
Request a Project
Project Help Wanted Ads
Publicize your Project
Submit Content

Get Informed

About java.net
Articles
Weblogs
News
Events
Also in Java Today
java.net Online Books
java.net Archives

Get Connected

java.net Forums
Wiki and Javapedia
People, Partners, and Jobs
Java User Groups
RSS Feeds

Search

Web and Projects: »
Online Books: »
Advanced Search

Home | Changes | Index | Search | Go

Project Wonderland v0.5: Creating a simple shape (Part 1)

by Jordan Slott (jslott@dev.java.net)

Purpose

In this tutorial, you will begin to learn how to create a new type of **Cell** for Project Wonderland. A Cell is a 3D volume; custom Cell types are the primary means developers extend Wonderland, allowing them to render custom graphics and interact with users.

This is the first in a series of four tutorials. In this first tutorial, you will learn how to write the essential classes that comprise a new Cell and how to create instances of the new Cell in-world via the Cell Palette. You will also learn how to package the Cell code into a module and deploy it to your Wonderland server.

This tutorial is designed for the Project Wonderland v0.5 User Preview 2.

In this tutorial you will develop the "shape" module. You can find the entire source code for this module, including code for future tutorials in the "unstable" section of the Project Wonderland modules workspace. For instructions on downloading this workspace, see Download, Build and Deploy Project Wonderland v0.5 Modules.

Expected Duration: 60 minutes

Prerequisites

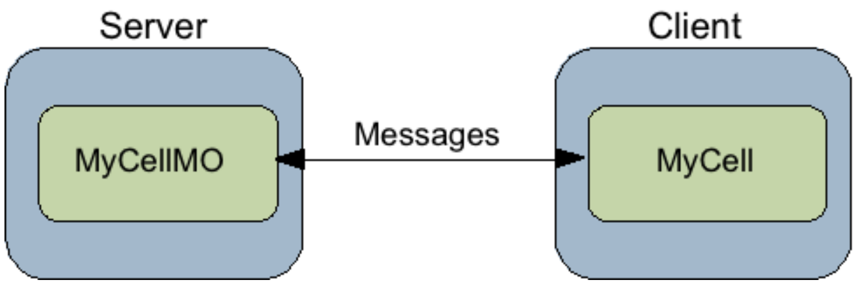
Before completing this tutorial, you should have already completed the following:

- Download, Configure, Build and Run from the Wonderland v0.5 Source
- Download, Build and Deploy Project Wonderland v0.5 Modules
- Wonderland Web-Based Administration Guide
- Project Wonderland: Working with Modules

In these tutorials you learned how to download and compile the Wonderland source code, run the Wonderland server, compile an example module project, and install the module into your Wonderland server.

Cell Architecture

The basic 3D visual building block inside Wonderland are *Cells*, comprised of a server piece and a client piece (Figure 1).



The server piece consists of a single class (MyCellMO, Figure 1) that maintains the shared state of the world across all client participants. It has the suffix 'MO' to indicate that it is a special object that participates in the Project Darkstar gaming engine mechanism. The client-side piece (MyCell, Figure 1) is responsible for rendering the Cell in-world either by loading an art resource or by drawing objects directly using the jMonkeyEngine 3D APIs.

Shape Cell: A New Kind of Cell

In this tutorial, you will learn how to create a new kind of Cell that draws a 3D shape in-world and register it with the Cell Palette mechanism.

Setting up your module project properties file

First, download an empty, sample module project : (.tar.gz (Linux/UNIX/Mac OSX) or .zip (Windows)).

1. Extract the archive file onto your local disk. Rename the directory to **shape-tutorial-module**.
2. Edit the **my.module.properties** file and make the following changes:
    - Change the value of **module.name** to **shape-tutorial-module**
    - Change the value of **module.description** to some meaningful description
    - If incorrect, modify the value of **wonderland.dir** to point to your Wonderland source code directory

- Change the value of **module.plugin.src** to **org/jdesktop/wonderland/modules/shape**

Your **my.module.properties** file should look something like this:

```
########################################################################################
# my.module.properties - Property files for Wonderland module projects
#
# Customize the properties in this file for your module project. A Wonderland module cor
# a collection art, plugins, and WFSs, etc.
########################################################################################

#
# Property: module.name (required)
# The unique name of the module
#
module.name=shape-tutorial-module

#
# Property: module.description (optional)
# A textual description of the module
#
module.description=A simple shape cell type module for Wonderland

#
# Property: wonderland.dir (required)
# The location of the Wonderland source
#
wonderland.dir=../../wonderland

#
# Property: module.plugin.src (optional)
# Beneath src/classes/, where is the module code located (common/, client/, server/)
#
module.plugin.src=org/jdesktop/wonderland/modules/shape
```

Loading the module project in Netbeans

In this tutorial, you'll use the Netbeans IDE to author the Cell code. Feel free to use your favorite IDE or command-line tools, although the specific steps may be slightly different. Once you have edited your **my.module.properties** file, open the project in Netbeans:

1. Start the Netbeans IDE
2. From the main menu select File -> Open Project...
3. Navigate to and select your **shape-tutorial-module/** directory
4. Click Open Project

The project should appear under the Projects tab in the upper-left of the IDE window. First you want to name the module project properly:

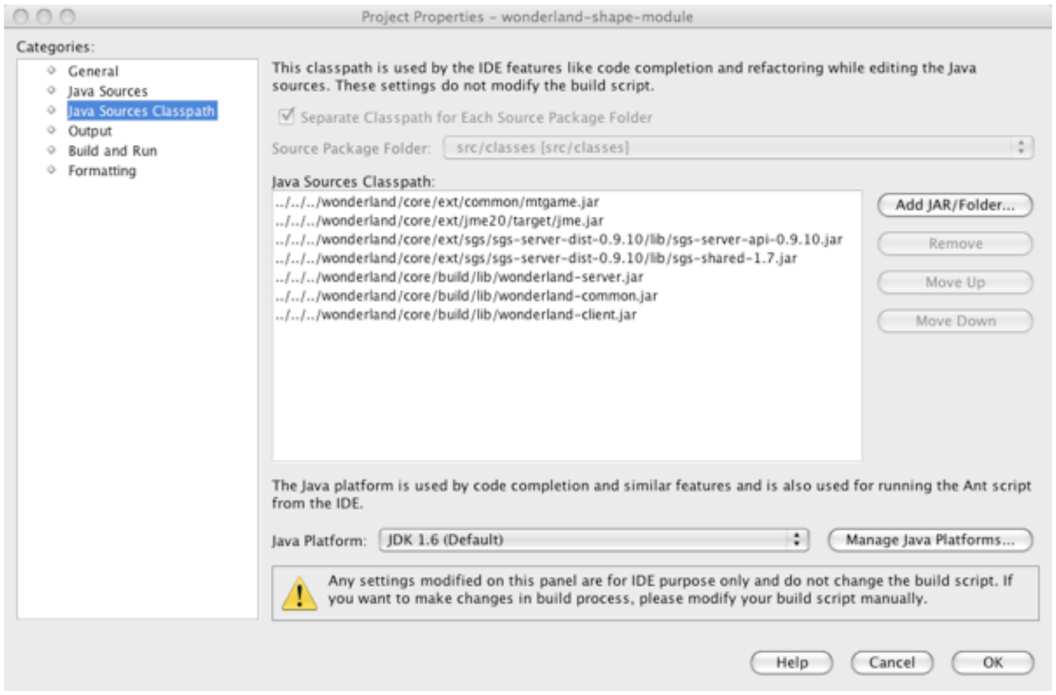
1. Right-click on the **wonderland-empty-module** project name and select Rename... from the menu. Enter a new name of **wonderland-shape-module**. There is no need to rename the project folder.

Next, you must set up your project so that it includes the proper JARs from the Project Wonderland workspace. Your Wonderland source should be compiled before completing this step.

1. Right-click on the **wonderland-shape-module** project and select Properties from the menu
2. Select Java Sources Classpath under the Categories column
3. If the existing entries are not correct, then select all of them and click Remove
4. Then re-add each JAR by clicking Add JAR/Folder and selecting the following files, if ${wonderland.dir} is where you Wonderland source installation is located:
   - ${wonderland.dir}/core/ext/common/mtgame.jar
   - ${wonderland.dir}/core/ext/jme20/target/jme.jar
   - ${wonderland.dir}/core/build/lib/wonderland-client.jar
   - ${wonderland.dir}/core/build/lib/wonderland-common.jar
   - ${wonderland.dir}/core/build/lib/wonderland-server.jar
   - ${wonderland.dir}/core/ext/sgs/sgs-server-dist-0.9.10/lib/sgs-shared-1.7.jar
   - ${wonderland.dir}/core/ext/sgs/sgs-server-dist-0.9.10/lib/sgs-server-api-0.9.10.jar
5. When finished Click OK

Note: This process can be expedited by manually editing the **project.xml** file, located in **{shape-module directory}/nbproject**. For example, if you want to replace every instance of "/wonderland/trunk/" with only "/wonderland/", this can be achieved using a simple search-and-replace action.

Your dialog box named **Project Properties - wonderland-shape-module** should look like this (Click on the image to see a full-sized version):



Project Properties - wonderland-shape-module

Categories:
- General
- Java Sources
- Java Sources Classpath
- Output
- Build and Run
- Formatting

This classpath is used by the IDE features like code completion and refactoring while editing the Java sources. These settings do not modify the build script.

☑ Separate Classpath for Each Source Package Folder

Source Package Folder: src/classes [src/classes]

Java Sources Classpath:

```
../../../wonderland/core/ext/common/mtgame.jar
../../../wonderland/core/ext/jme20/target/jme.jar
../../../wonderland/core/ext/sgs/sgs-server-dist-0.9.10/lib/sgs-server-api-0.9.10.jar
../../../wonderland/core/ext/sgs/sgs-server-dist-0.9.10/lib/sgs-shared-1.7.jar
../../../wonderland/core/build/lib/wonderland-server.jar
../../../wonderland/core/build/lib/wonderland-common.jar
../../../wonderland/core/build/lib/wonderland-client.jar
```

The Java platform is used by code completion and similar features and is also used for running the Ant script from the IDE.

Java Platform: JDK 1.6 (Default)

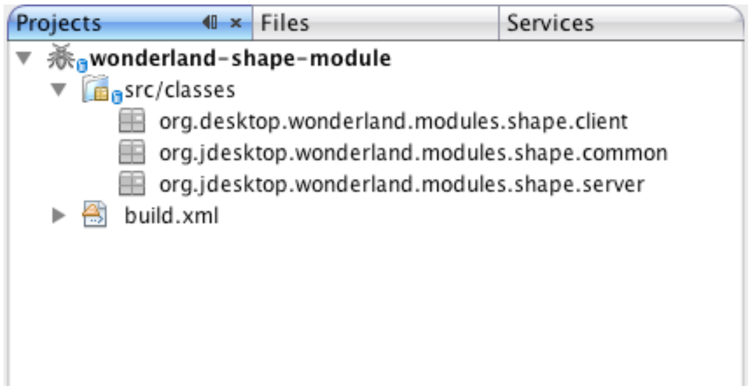
Any settings modified on this panel are for IDE purpose only and do not change the build script. If you want to make changes in build process, please modify your build script manually.

Source package architecture

Once you have your project and classpath all set up, you can create the directories in which your source code will reside. The source code for Cells are broken up into three packages: **client**, **common**, and **server**. Classes in the **client** package are compiled into the Wonderland client, classes in the **server** package are compiled into the Wonderland server, and classes in the **common** package are compiled into both the client and server.

1. Right-click on the **src/classes** folder and select New -> Java Package... Enter "org.jdesktop.wonderland.modules.shape.client" in the File Name text box and click Finish.
2. Right-click on the **src/classes** folder and select New -> Java Package... Enter "org.jdesktop.wonderland.modules.shape.common" in the File Name text box and click Finish.
3. Right-click on the **src/classes** folder and select New -> Java Package... Enter "org.jdesktop.wonderland.modules.shape.server" in the File Name text box and click Finish.

Note that you will also need a package called **org.jdesktop.wonderland.modules.shape.client.jme.cellrenderer**. However, a quirky behavior of Netbeans will not let you create both the **client.jme.cellrenderer** and **client** packages with no classes in either of them. So, you'll just create the **client.jme.cellrenderer** package below in a bit. When finished, your Projects pane should look like the following:



Projects
Files
Services
wonderland-shape-module
src/classes
org.desktop.wonderland.modules.shape.client
org.jdesktop.wonderland.modules.shape.common
org.jdesktop.wonderland.modules.shape.server
build.xml

The ShapeCellMO server class

The first class you will create is the server-side class representing the Cell, ShapeCellMO. The purpose of the server-side class is to manage the shared state of the Cell for all clients. To help manage that state, and the individual, asynchronous updates to the state from clients, the ShapeCellMO class participates in the Project Darkstar transactional gaming infrastructure. In this first example, however, the ShapeCellMO does not maintain shared state for different clients, and will not make use of the Darkstar transaction mechanism. (Future tutorials, will however, exercise some of this functionality.)

1. Right-click on the **org.jdesktop.wonderland.modules.shape.server** package and select New -> Java Class... Name the class ShapeCellMO and click Finish.

Netbeans should create a skeleton of a class that looks something like this:

```
package org.jdesktop.wonderland.modules.shape.server;

public class ShapeCellMO {
}
```

First, copy the necessary import statements into your Java class directly beneath the package statement:

```
import org.jdesktop.wonderland.common.cell.ClientCapabilities;
import org.jdesktop.wonderland.common.cell.state.CellClientState;
import org.jdesktop.wonderland.common.cell.state.CellServerState;
import org.jdesktop.wonderland.modules.shape.common.ShapeCellClientState;
import org.jdesktop.wonderland.modules.shape.common.ShapeCellServerState;
import org.jdesktop.wonderland.server.cell.CellMO;
import org.jdesktop.wonderland.server.comms.WonderlandClientID;
```

Next, modify the definition of your Java class to be:

```
public class ShapeCellMO extends CellMO {
```

This class extends CellMO that resides in the **org.jdesktop.wonderland.server.cell** package and is the base class for all server-side Cells. The ShapeCellMO class is a "managed object" in the context of Project Darkstar (because the CellMO class implements the ManagedObject interface).

For good measure, go ahead and define the default constructor for this class:

```
    public ShapeCellMO() {
    }
```

The ShapeCellMO class maintains one piece of configurable information: a String describing the shape of the object to draw: either BOX or SPHERE. You will store this String as a field in the ShapeCellMO class:

```
       private String shapeType = null;
```

The CellMO class declares one abstract method, *getClientCellClassName()*, that you must implement. This method should return the fully-qualified class name of the Cell's client-side counterpart.

```
    @Override
    public String getClientCellClassName(WonderlandClientID clientID, ClientCapabilities
        return "org.jdesktop.wonderland.modules.shape.client.ShapeCell";
    }
```

You will also need to override three methods that help set and get the server and client state of the Cell. You'll learn about the meaning of the "server" and "client" state of the Cell below. For now, just implement these three methods; their meaning will be become clearer shortly.

First override the **setServerState()** method to take a class that represents the server-side state of the Cell. In our case, the server-side state will just consist of the String shape type. Various parts of the system will call **setServerState()** on your Cell: to initialize the state of your Cell or to update the state of your Cell.

```
    @Override
    public void setServerState(CellServerState state) {
        super.setServerState(state);
        this.shapeType = ((ShapeCellServerState)state).getShapeType();
    }
```

The **setServerState()** method takes an argument of type CellServerState -- the base class for all server state objects. You will define a subclass of CellServerState (called ShapeCellServerState) below. For now, just fetch the shape type from your server state object and set the member variable in the ShapeCellMO class. Since the CellMO class defines its own **setServerState()** method, it is really important to use the @Override annotation--that'll flag any typos you might make when defining the signature of your **setServerState()** method. Also, be sure to invoke **super.setServerState()**: the CellMO super class does important things here (e.g. to set the position, rotation, and scaling of the Cell).

The next method you will define, **getServerState()**, is the complement to the **setServerState()** method. It fetches information stored in your ShapeCellMO class and stores it in the Cell's server state class.

```
    @Override
    public CellServerState getServerState(CellServerState state) {
        if (state == null) {
            state = new ShapeCellServerState();
        }
        ((ShapeCellServerState)state).setShapeType(shapeType);
        return super.getServerState(state);
    }
```

This method is passed an argument of type CellServerState. Most often, this argument is null, and you should create an instance of your server-state object. A CellServerState argument is passed in to allow your ShapeCellMO class to be subclassed by other Cell types: in this way, the subclassing Cell will write its own **getServerState()** method and create its own server-side state object (which must also extend your server-side state object). If you receive a non-null argument, it means a subclass has created a more specific instance of the Cell server-state object. Of course, if the subclass does not also subclass your server state object, you'll receive a ClassCastExceptionon the **setShapeType()** line. If the argument is not-null, you are more then welcome to make sure it is also an instance of your ShapeCellServerState object, but it's not something we bothered with here.

Two other notes: It is important to **only** create a new instance of your server-side state object (ShapeCellServerState) in the if-null clause, to handle the subclassing case discussed above where you may receive a non-null argument. Also, make sure you invoke **super.getServerState()** so that the CellMO class can set important information on the Cell server-state class.

Finally, you need to implement a method that returns the client-state of the Cell, based upon the current server-side state of the Cell.

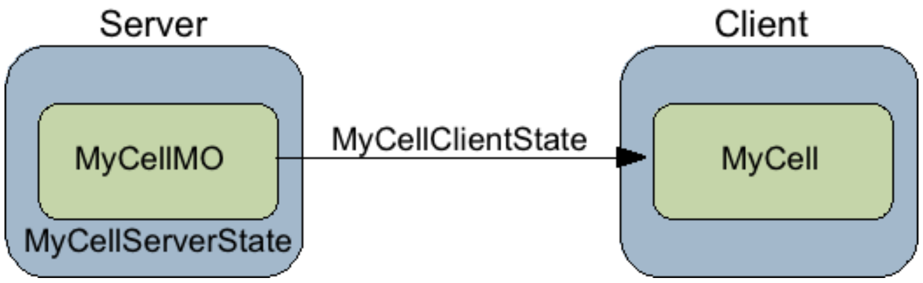
```
    @Override
    public CellClientState getClientState(CellClientState cellClientState, WonderlandCli
        if (cellClientState == null) {
            cellClientState = new ShapeCellClientState();
        }
        ((ShapeCellClientState)cellClientState).setShapeType(shapeType);
        return super.getClientState(cellClientState, clientID, capabilities);
    }
```

The pattern of **getClientState()** is very similar to that of **getServerState()**. There are a couple more arguments (clientID and capabilities) which you do not need to worry about now. Note in this case, the client state is exactly the same as the server state: the type of the shape. This does not always need to be the case: we maintain the distinction to cleanly separate these concepts and to allow more advanced usages of client and server state.

ShapeCellServerState versus ShapeCellClientState

The ShapeCellMO class used two different, but related objects: ShapeCellServerState and ShapeCellClientState. Each stores some sort of information describing the Cell, but for different purposes (Figure 2). The ShapeCellServerState class extends the CellServerState class and represents the information used to properly create the server-side Cell class. This class is also used to export the state of the Cell for backup (called "snapshots" in Wonderland) and exporting purposes. Each Cell typically has a "server state" class, and by convention, ends with "ServerState" in its class name.

The ShapeCellClientState class extends the CellClientState class and represents the information that is passed from the server to tell the client how to configure itself. Often times, the "ServerState" class and "ClientState" class contain the same information, but not always. The "ClientState" class is serialized across to each client for each Cell when it connects.



The ShapeCellServerState common class

Next, you will implement the ShapeCellServerState class that is compiled both into the server-side and client-side code, and therefore belongs in the "common" package. (Even though this class is used for "server state" information on the server, there exists client-side tools that require this class too).

This class is responsible for reading and parsing the XML Cell file within WFS. The Wonderland File System (WFS) is a series of XML files on disk that describe the layout and state of a world on disk. In previous releases, it was used as the main means to configure a world; in version 0.5, creating and maintaining the state of the world is far more dynamic and done via tools built into the Wonderland client. We will talk about one of these tools below: the Cell Palette. Yet, WFS still plays an important role in Wonderland: it is used to make back-ups of the world state (called 'snapshots') and is also used as an import/export format to share world state with other servers. It is the responsibility of the Cell server-state class to properly map how its state information is mapped to XML.

For XML parsing, it uses JAXB: Java Architecture for XML Binding. Previous versions of Wonderland (v0.3-v0.4) uses the Java Bean XML encoding and decoding mechanism: although it was simple to write a class, the resulting XML was needlessly complicated. In v0.5, Wonderland uses JAXB to generate neat XML files within WFS. The trade-off is that the "ServerState" class becomes slightly more complicated.

1. Right-click on the **common** package and select New -> Java Class... Name the class ShapeCellServerState and click Finish.

Implement this class as follows:

```
package org.jdesktop.wonderland.modules.shape.common;

import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlRootElement;
import javax.xml.bind.annotation.XmlTransient;
import org.jdesktop.wonderland.common.cell.state.CellServerState;
import org.jdesktop.wonderland.common.cell.state.annotation.ServerState;
```

```
@XmlRootElement(name="shape-cell")
@ServerState
public class ShapeCellServerState extends CellServerState {

    @XmlElement(name="shape-type")
    private String shapeType = "BOX";

    public ShapeCellServerState() {
    }

    @XmlTransient public String getShapeType() { return this.shapeType; }
    public void setShapeType(String shapeType) { this.shapeType = shapeType; }

    @Override
    public String getServerClassName() {
        return "org.jdesktop.wonderland.modules.shape.server.ShapeCellMO";
    }
}
```

This class stores the type of the shape as a String and has a default constructor and the standard setter and getter methods. If you wish to extend the functionality of the shape Cell type (e.g. by including a radius or color configurable parameters), you would include these new parameters in this class, for example.

Note it also contains several annotations: @XmlRootElement, @XmlElement, and @XmlTransient. These are all annotations provided by JAXB. All "ServerState" classes must have an @XmlRootElement annotation--the annotation defines the root element in the WFS Cell's XML file, and the annotation always appears right before the class definition. That is, the server state information for the Cell will be contained within the <shape-cell>...</shape-cell> tags.

Next, you'll want to specify how each field in the ShapeCellServerState class appears in the XML file. Typically, fields appear as XML elements, that is: <shapetype>SPHERE</shapetype>. To achieve this, simply use an @XmlElement annotation before the field declaration (along with the desired element name in XML), and use an @XmlTransient annotation before the getter method for this field.

In your ShapeCellServerState class you must also implement the **getServerClassName()** method to return the fully-qualified name of your server-side Cell class, in this case **org.jdesktop.wonderland.modules.shape.server.ShapeCellMO**. When the system encounters an XML file on disk, it uses the root tag to determine which "ServerState" class to use to decode the XML file (based upon all classes with an @XmlRootElement annotation it knows about), from the "ServerState" class it discovers which server-side Cell class to instantiate, from the server-side Cell class, it discovers which client-side Cell class to instantiate.

There is one more step before we are finished with this class, a step you have already taken above, but it is worth an explanation now. One complication of the JAXB mechanism is that Wonderland must know the names of all "ServerState" classes with the @XmlRootElement annotations. This is accomplished by simply adding the @ServerState annotation at the top of your class.

The ShapeCellClientState common class

Next, you will implement the ShapeCellClientState class that is compiled both into the server-side and client-side code, and therefore belongs in the "common" package. This class contains similar information as the ShapeCellServerState class--in fact, it is a bit redundant, but we figured the overhead of two separate classes was not all that great. Since it is serialized from the server to each client as a Java object, you do not need to worry about JAXB annotations.

1. Right-click on the **common** package and select New -> Java Class... Name the class ShapeCellClientState and click Finish.

Implement this class as follows:

```
package org.jdesktop.wonderland.modules.shape.common;

import org.jdesktop.wonderland.common.cell.state.CellClientState;

public class ShapeCellClientState extends CellClientState {
    private String shapeType = null;

    public ShapeCellClientState() {
    }

    public String getShapeType() {
        return shapeType;
    }

    public void setShapeType(String shapeType) {
        this.shapeType = shapeType;
    }
}
```

The ShapeCell client class

Finally, you will implement the ShapeCell client-side class that draws the shape in-world.

1. Right-click on the **client** package and select New -> Java Class... Name the class ShapeCell and click Finish.

Implement this class as follows:

```
package org.jdesktop.wonderland.modules.shape.client;

import org.jdesktop.wonderland.common.cell.state.CellClientState;
import org.jdesktop.wonderland.modules.shape.client.jme.cellrenderer.ShapeCellRenderer;
import org.jdesktop.wonderland.client.cell.Cell;
import org.jdesktop.wonderland.client.cell.Cell.RendererType;
import org.jdesktop.wonderland.client.cell.CellCache;
import org.jdesktop.wonderland.client.cell.CellRenderer;
import org.jdesktop.wonderland.common.cell.CellID;
import org.jdesktop.wonderland.modules.shape.common.ShapeCellClientState;

public class ShapeCell extends Cell {

    private String shapeType = null;
    private ShapeCellRenderer renderer = null;

    public ShapeCell(CellID cellID, CellCache cellCache) {
        super(cellID, cellCache);
    }

    @Override
    public void setClientState(CellClientState state) {
        super.setClientState(state);
        this.shapeType = ((ShapeCellClientState)state).getShapeType();
    }

    public String getShapeType() {
        return this.shapeType;
    }

    @Override
    protected CellRenderer createCellRenderer(RendererType rendererType) {
        if (rendererType == RendererType.RENDERER&#95;JME) {
            this.renderer = new ShapeCellRenderer(this);
            return this.renderer;
        }
        else {
            return super.createCellRenderer(rendererType);
        }
    }
}
```

The ShapeCell class extends the Cell class that provides the basic functionality of all client-side Cells. Each sublcass of class Cell must override the **setClientState()** method that takes the set of parameters configured on the server side and draws the Cell in-world. In this case, the CellClientState object is an instance of the ShapeCellClientState class that we defined above. The **setClientState()** method simply stores the type of shape communicated from the server.

You'll note that this client-side class does not actually draw the shape--that is left to the renderer of the Cell. In Wonderland v0.5, the client-side Cells consist of a collection of "components", each component performs a certain task. For example, a Cell may have a "moveable" component that lets the Cell move, or a "treatment" component that plays audio. Much more will be said about the client-side component model in future tutorials.

All Cells have a "renderer" component that is responsible for actually drawing the Cell. Currently, our Cell only supports renderers that work with jMonkeyEngine (jME). Next, you'll implement the renderer for this Cell.

1. Right-click on the **src/classes** folder and select New -> Java Package... Enter "org.jdesktop.wonderland.modules.shape.client.jme.cellrenderer" in the File Name text box and click Finish
2. Right-click on the **cellrenderer** package and select New -> Java Class... Name the class ShapeCellRenderer and click Finish

Implement this class as follows:

```
package org.jdesktop.wonderland.modules.shape.client.jme.cellrenderer;

import com.jme.bounding.BoundingBox;
import com.jme.math.Vector3f;
import com.jme.scene.Node;
import com.jme.scene.TriMesh;
import com.jme.scene.shape.Box;
import com.jme.scene.shape.Sphere;
import org.jdesktop.wonderland.client.cell.Cell;
import org.jdesktop.mtgame.Entity;
import org.jdesktop.wonderland.client.jme.cellrenderer.BasicRenderer;
import org.jdesktop.wonderland.modules.shape.client.ShapeCell;

public class ShapeCellRenderer extends BasicRenderer {

    private Node node = null;

    public ShapeCellRenderer(Cell cell) {
        super(cell);
    }

    private TriMesh getShapeMesh(String name, String shapeType) {
        TriMesh mesh = null;
        if (shapeType != null && shapeType.equals("BOX") == true) {
            mesh = new Box(name, new Vector3f(), 2, 2, 2);
        }
        else if (shapeType != null && shapeType.equals("SPHERE") == true) {
            mesh = new Sphere(name, new Vector3f(), 25, 25, 2);
        }
        else {
            logger.warning("Unsupported Shape type " +cell.getLocalBounds().getClass().g
        }
        return mesh;
    }

    protected Node createSceneGraph(Entity entity) {
        String name = cell.getCellID().toString();
        String shapeType = ((ShapeCell)cell).getShapeType();

        TriMesh mesh = this.getShapeMesh(name, shapeType);
        if (mesh == null) {
          node = new Node();
          return node;
        }

        node = new Node();
        node.attachChild(mesh);
        node.setModelBound(new BoundingBox());
        node.updateModelBound();
        node.setName("Cell&#95;"+cell.getCellID()+":"+cell.getName());

        return node;
    }
}
```

The primary method in this class is **createSceneGraph()** that is responsible for creating the scene graph for you Cell. It returns an object of type **com.jme.scene.Node**. It is beyond the scope of this tutorial to describe the details of the jME API in detail (but we've posted some tips) -- this method simply creates either a Box or Sphere and places it in the jME scene graph Node. We keep the Node object around for later tutorials; the **getShapeMesh()** method will also come in handy in the future, it simply creates either a new sphere or cube shape mesh depending upon the arguments passed to it.

Entity Objects

You'll notice that the **createSceneGraph()** method takes a single argument of type **org.jdesktop.mtgame.Entity**. An Entity is a concept introduced in Project Wonderland v0.5 and is defined by the multi-threaded graphics engine in Project Wonderland. An Entity represents some "object" in the world. The definition of an "object" is somewhat loose -- it does not strictly have to be a single shape object. Rather an Entity can represent a compound shape, such as an avatar. An Entity is the smallest, indivisible thing where certain elements of the virtual world are handled. Keyboard and mouse events are handled on the Entity level, for example. The visual aspects of an Entity are created by attaching a jME scene graph to the Entity.

All Cells have at least one Entity: the "root" entity is created for you and passed into the **createSceneGraph()** method. You can create sub-Entities for the root Entity within your Cell, but most often not. The **createSceneGraph()** method returns a Node object that is attached to the root Entity for you.

Defining a Cell Factory for the Shape Cell

There's one more class you'll need to write and is the first step towards dynamically creating Cells in-world. In order for your Cell to appear in this palette, you just need to write a relatively simple "factory" class.

1. Right-click on the **client** package and select New -> Java Class... Name the class ShapeCellFactory and click Finish

Define this class as follows:

```
package org.jdesktop.wonderland.modules.shape.client;

import java.awt.Image;
import java.util.Properties;
import org.jdesktop.wonderland.client.cell.registry.annotation.CellFactory;
import org.jdesktop.wonderland.client.cell.registry.spi.CellFactorySPI;
import org.jdesktop.wonderland.common.cell.state.CellServerState;
import org.jdesktop.wonderland.modules.shape.common.ShapeCellServerState;

@CellFactory
public class ShapeCellFactory implements CellFactorySPI {

    public String[] getExtensions() {
        return new String[] {};
    }

    public <T extends CellServerState> T getDefaultCellServerState(Properties props) {
        return (T)new ShapeCellServerState();
    }

    public String getDisplayName() {
        return "Shape Tutorial";
    }

    public Image getPreviewImage() {
        return null;
   }
}
```

This class implements the **CellFactorySPI** interface. There are several key methods: the **getDisplayName()** method returns the name to appear in the Cell Palette. You can also associate a 128x128 preview icon with your Cell, using the **getPreviewImage()** method, although we return null in this simple case. The **getExtensions()** method returns the array of file extensions of media that your Cell can display. Your Cell does not support any media, so simply return an empty array (don't return null here, just an empty array). (An example of a kind of Cell that would return a value for the **getExtensions()** method is a "PDF Viewer" Cell that would indicate it supports files with a .pdf extension. Currently this method is not used anywhere in the system, but it will be in the future).

The **getDefaultCellServerState()** method is central to creating Cells via the Cell Palette. The system will invoke this method when you wish to create a Cell and send the object to the server in a message. From this message, the server will know how to create your Cell. Here, you will just return an instance of the **ShapeCellServerState** object; you can use this method to set default parameters in **ShapeCellServerState** too.

An important part of this class is the @CellFactory annotation: it is this annotation that tells the system that your **ShapeCellFactory** should appear in the Cell Palette.

Compiling, Installing, and Running the Shape Module

To install your new module in Wonderland, follow the directions in Project Wonderland: Working with Modules, starting with the "Compiling the shape module project" heading. Note you can also invoke **ant deploy** from the command-line or use the *Deploy To Server* action if you right-click on the project in the Netbeans Project viewer.

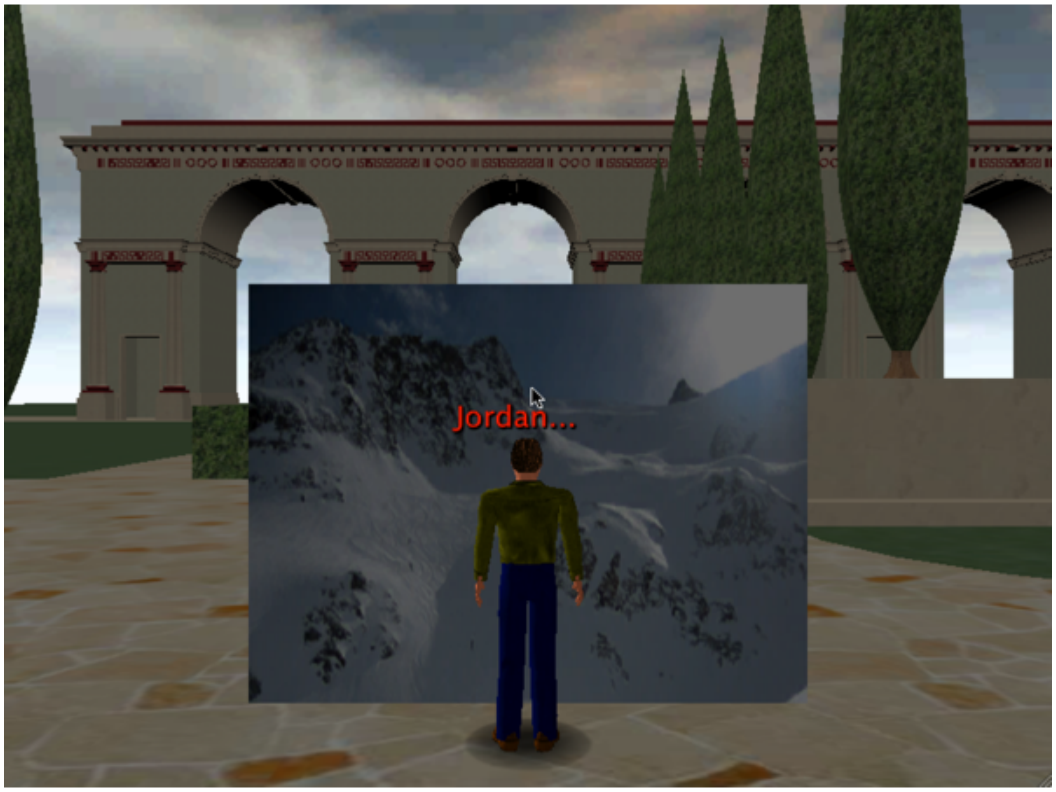
Once you have installed your shape module on the server, and confirmed it has been installed via the Web Administration UI, please run your client via Java Webstart or the command-line (via **ant run** in the core/ directory).

Creating a New Cell using the Cell Palette

Once you have run your client:

1. Select Insert -> Object... from the main menu
2. In the list of modules, select "Shape Tutorial" and click the Create button

You should now see the sample Cell (a BOX) in the world directly in front of you. Your client should look something like this: (Click on the image to view a full-sized version). Note that this screenshot was taken after the end of the four-part tutorial series; your shape should appear gray rather than textured with a mountain scene.



Extra Credit

If you would like to extend the functionality of this new Cell type, you may make the radius of the object or its color configurable. These changes would involve both the client and server class, as well as the ShapeCellServerState class. You may also add support for other types of shapes (e.g. cones, cylinders). This basic Cell type can be used as a template for any Cell that draws into the world using jMonkeyEngine primitives--feel free to experiment!

Next Steps

In the next tutorial, you will learn how to receive notification of events (e.g. mouse clicks) for your Cell. In future tutorials you will also texture your shape with an image and manage and update the state of your Cell shared among many clients.

Part 2 - Accepting Mouse Input in the Shape Cell
Part 3 - Texturing the Shape Cell Type
Part 4 - Synchronizing the State of the Shape Cell Type