HW5: Agents 2

CSE1102 Spring 2015
Jeffrey A. Meunier
University of Connecticut

1. Introduction

In this project you will extend the *Agents and Spaces* assignment to include two new features:

- Each space will have an image associated with it that will be shown when the agent moves into that space.
- The information about spaces and portals will be stored in a configuration file that is external to the Java program itself. That means that it no longer requires Java programming skill to create a new set of spaces and portals.

Also, you will be given the opportunity to create some images of your own to use as visual representations of the spaces that you create.

Those of you familiar with games can think of the Java program as the game engine, and the configuration file as a single level. Right now only one configuration file is allowed.

2. Due Date

This project is due by midnight at the end of the day on **Sunday, March 29, 2015**. A penalty of 20% per day will be deducted from your grade, starting at 12:00:01am. See this link for additional information.

3. Value

This program is worth a maximum of 20 points. See the grading rubric at the end for a breakdown of the values of different parts of this project.

4. Objectives

Here are some objectives of this assignment:

- **for** loops
- Data conversion from textual specification into interconnected objects.
- A bit of GUI programming.
- Using some built-in classes, even if you don't know how they work. Think about it: you can use anything as long as you know how the interface to that thing works. You don't need to know how a car or a computer works in order to use it. The same is true for Java classes.

- Using external jar files in Eclipse.

5. Background

This is a reasonable place to mention this: please place comment blocks in ALL your class files, not just the **Main** class file.

5.1. Private methods

In this assignment I have you create several private methods. This is unusual, because usually methods are public. The reason I have you make the methods private is because they're not needed outside the class in which they are created. If they can be made private, then they should be made private. Don't expose internal features of a class unless you know that you should, meaning a variable or method should be private unless you know that it should be public.

5.2. INI files

I have written a Java library that you will use for this project (a library is a collection of classes that is not by itself a complete program). It allows your program to load configuration data from an external file. The the data in the file follows the INI format that Microsoft Windows uses. You can read Wikipedia's article about INI files here.

The format of the data in the file is simple. A file has sections. Each section has a header in square brackets, and under each section there are properties (or *key* = *value* pairs), like this:

```
[section]
key = value
```

The **Ini** class library loads an INI file from disk and stores it in memory, and builds a logical structure from the sections in the file. You can query the **Ini** structure for what the section names are, what properties are found in a specific section, what the value is for a given key in a given section. You can also add or change properties and have the new INI file written back to disk.

The data stored in the INI file is rudimentary, but it contains enough information to make your simulation configurable through this INI file, which is separate from the program class files. In this project you will write a **ConfigLoader** class that will convert the INI file into a collection of spaces and portals, just like what happens in your main method now.

You can even send your INI file to a friend, and your friend can run your simulation without re-compiling the Java program.

This is what my main method used to look like:

```
  Space classroom = new Space();
  classroom.setName("classroom");
  classroom.setDescription("a large lecture hall");
  Space sidewalk = new Space();
  sidewalk.setName("sidewalk");
  sidewalk.setDescription("a plain concrete sidewalk with weeds growing
through the cracks");
  Portal door = new Portal();
  door.setName("door");
  door.setDirection("outside");
  door.setDestination(sidewalk);
  classroom.setPortal(door);
  Agent student = new Agent();
  student.setName("Harry Potter");
  student.setLocation(classroom);
```

But now it looks like this:

```
  File configFile = new File("data", "config.ini");
  ConfigLoader cl = new ConfigLoader(configFile);
  Agent a = cl.load();
  CommandInterpreter.run(a);
```

This is what my **config.ini** file looks like:

```
  [spaces]
  classroom = a large lecture hall
  sidewalk = a plain concrete sidewalk with weeds growing through the cracks
  [images]
  classroom = classroom.jpg
  sidewalk = sidewalk.jpg
  [portals]
  door = outside
  [exits]
  classroom = door
  [destinations]
  door = sidewalk
  [agents]
  Harry Potter = classroom
```

```
[start]
agent = Harry Potter
```

This is the format of each section:

- **spaces**: list each space in the form ***spaceName = spaceDescription***
- **portals**: list each portal in the form ***portalName = destinationDescription***
- **exits**: this connects spaces to their exit portals, the form is ***spaceName = portalName***
- **destinations**: this connects portals to their destinations, the form is ***portalName = destinationSpaceName***
- **agents**: list each agent in the form ***agentName = startSpaceName***
- **start**: list the starting agent in the form ***agent = startingAgentName***

Note that for each exit in the **exits** section, the left hand side must refer to a space listed in the **spaces** section and the right hand side must be a portal listed in the **portals** section.

For each destination in the **destinations** section, the left hand side must be a portal in the **portals** section and the right hand side must be a space in the **spaces** section.

The start agent must be an agent listed in the **agents** section.

I know what you're thinking, but right now the simulation does not handle more than one agent. ;)

Getting values from the Ini instance

If you call the **get** method on the **Ini** instance, it will return the value (right hand side) associated with a specific key (left hand side). For instance, have a look at the **spaces** section:

```
[spaces]
classroom = a large lecture hall
sidewalk = a plain concrete sidewalk with weeds growing through the cracks
```

Assume that you have the Ini instance stored in a variable called **ini**. The **get** method is used to retrieve the value associated with a key:

```
File   iniFile = new File("data", "config.ini");
Ini    ini     = new Ini(iniFile); // loads the ini file

String section = "spaces";
String key     = "classroom";
String value   = ini.get(section, key);
```

The **value** variable will then contain the string "a large lecture hall".

Note that the **Ini** instance stores only strings. In this assignment you will need to analyze the strings to build instances of **Space**, **Portal**, and **Agent**, and also connect them together.

6. Assignment

Read through all of the subsections in this section before you start, to make sure you understand how you must proceed with the assignment.

6.1. Copy HW 3

Summary

This project extends the previous assignment.

Do this

Start Eclipse and make a copy of your HW 3 project. If your HW 3 project does not work, contact your TA to obtain a working copy of the project.

6.2. Modify Space class

Summary

It is not good programming to always rely on using a default constructor in a class, only to fill in the member variables later using setter methods. Instead, it is better to write a constructor that lets you supply more information when you create an instance.

Do this

Add a private member variable to hold the name of the image that is to be associated with this space. The name of the image will be a **String**. Eclipse will complain that this member variable is not used. That's fine for now. You will use it later.

Add a *getter* for this new member variable. Do not add a setter (once the file name has been stored in the instance, we do not ever need to change it).

Add a constructor to this class that has one parameter for each member variable. Inside the constructor, store the arguments into the member variables. I will refer to this as a full, simple constructor. It is *full* in that it has one parameter for each member variable, and it is *simple* in that all it does is store the arguments into the member variables.

The order in which you place the parameters in the parameter list is irrelevant, as is the order of the member variables in the class.

Change the **main** method so that it uses this new constructor for the **Space** class. For instance, for one **Space** instance I have these three statements in my **main** method:

```
Space classroom = new Space();
classroom.setName("classroom");
classroom.setDescription("a large lecture hall");
```

These should be replaced with this one method:

```
Space classroom = new Space("classroom", "a large lecture hall", null, null);
```

Be sure to remove the calls to the *setter* methods, as I have done. Do not actually delete the *setter* methods from the classes.

Notice that I don't yet know what the portal and the image name are, so I use **null** in those positions in the argument list. Later you will write a class that will provide all four arguments when it creates a Space instance.

Testing

After you change all the constructors, run the program again to be sure it works the same as it did before.

6.3. Modify Portal class

Summary

You will write a constructor for the **Portal** class as you did with the **Space** class.

Do this

Write a full, simple constructor for the **Portal** class (it should have 3 parameters).

Change the **main** method so that this new constructor is used, and that there are no setters called on your **Portal** instances.

You will still need to call the **setPortal** method on your **Space** instances.

Testing

Test your program again to be sure it still works the same way.

6.4. Modify Agent class

Summary

You will write a constructor for the **Agent** class as you did with the **Space** and **Portal**

classes.

Do this
Write a full, simple constructor for the **Agent** class (it should have 2 parameters).

Change the **main** method so that this new constructor is used, and that there are no setters called on your **Agent** instance.

After making this modification, the only *setter* methods called in the **main** method are the ones to set the portals of the spaces.

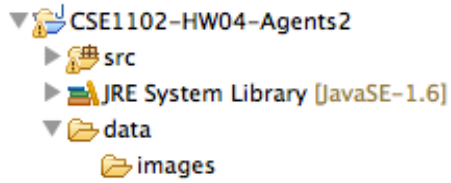
Testing
Test your program again to be sure it still works the same way.

6.5. Create a data folder
Summary
This project will require the use of several data files. These will be stored in a data folder in your project folder.

Do this
Right click on the project in Eclipse and select **New** > **Folder**. Make sure that the project is selected in the project list. Enter the folder name **data**. Click **Finish**.

Right click on the **data** folder and create a sub-folder named **images**.

Testing
Just have a look to make sure that your folder arrangement is similar to mine:



6.6. Store some images in that folder
Summary
You will incorporate some images into your project. Later you will add some methods that display the image associated with whatever space the agent is in.

Do this

Download these two image files and store them in the **data/images** folder in Eclipse.
https://dl.dropboxusercontent.com/u/2234170/courses/CSE1102/images/classroom.jpg
https://dl.dropboxusercontent.com/u/2234170/courses/CSE1102/images/sidewalk.jpg

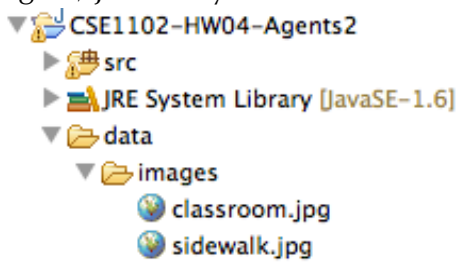
You can drag & drop the image files right into Eclipse, then choose the option to *copy* the images.

Go back into the main method and add the image names to the **Space** constructors. Do not supply the full path, only the name of the image file as it appears in the **data/images** folder. Here's one of the **Space** constructors in my program:

```
Space classroom = new Space("classroom", "a large lecture hall",
  "classroom.jpg", null);
```

Testing

Again, just verify that the files are in the folder (yes, it was previously homework 4):



6.7. Use the imagewindow jar file

Summary

I have written the classes that will do the work of displaying the images. All you need to do is install the jar file and then call the right methods at the right time.

Do this

Download this file and place it in your **ExternalLibs** folder in Eclipse.
https://dl.dropbox.com/u/2234170/courses/CSE1102/jars/imagewindow-1.0.jar
See section 5.3 in project 2 (Robot Grid) if you need instructions on how to do this.

PLEASE NOTE: Don't just include the jar file in the build path. Instead, you should set up a path variable for this called **IMAGEWINDOW**. That way when your TA loads your project into Eclipse, your project will refer to the variable, which will point to where the jar file is on his computer instead of where it is on your computer, which is undoubtedly in a different place. View this video to learn how to set up a variable for the imagewindow jar file.
https://dl.dropbox.com/u/2234170/courses/CSE1102/videos/EclipseVar.mov

After you set up the jar file, create an **ImageWindow** instance in the **main** method and store it in a variable (the class is in **jeff.imagewindow**). It should be very easy. There is only a default constructor for this class. Supply this variable as an argument to the **CommandInterpreter.run** method. This means that you must also add it as a parameter in the **CommandInterpreter.run** method. Do that now.

Write a **private static void** method in the **CommandInterpreter** class called **_showImage** (I like to use underscores with private methods, too). This method should have two parameters: an **ImageWindow** and an **Agent**. Get the name of the image associated with the agent's space, then display the image by using these three statements:

```
File imageDir = new File("data", "images");
File imageFile = new File(imageDir, imageName);
imageWindow.loadImage(imageFile);
```

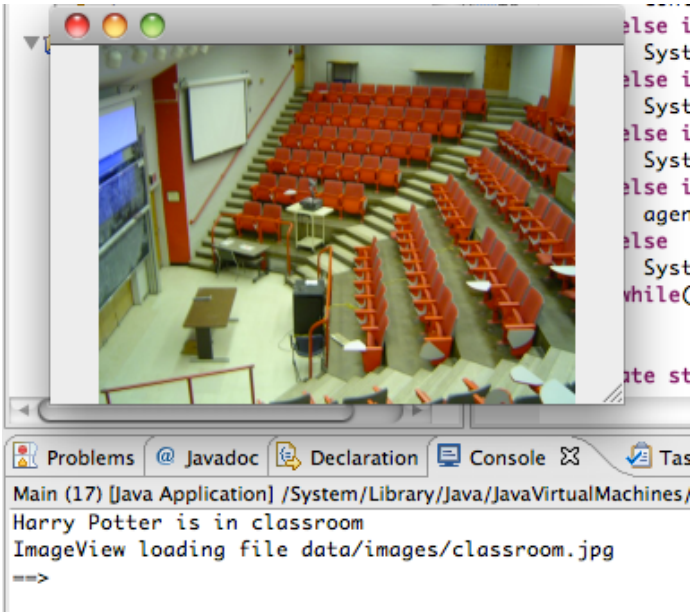
Where **imageName** is defined by you in this method, and **imageWindow** is the parameter.

Back in the run method, call the **_showImage** method just before the **do** loop starts.

In the part of the **if/else** ladder where you handle the "go" command, after you call the **usePortal** method, call the **_showImage** method the same way you did before the **do** loop. This will cause the image to change each time the agent moves.

Testing

Run the program. You should see your program running in the console, but an additional window should appear showing the image for the current space. This is what mine looks like:



Hmm, this is terribly inconvenient. Eclipse covers the image window each time you need to type in the console. We'll fix that next.

6.8. Use the textconsole jar file

Summary

I have written another jar file for you to use. This one places the console in a window that is separate from Eclipse. This way you can have the image window and the console window both on top of Eclipse at the same time.

The console that I have created provides these methods that you will can use:

- **void print(String)**
- **void println(String)**
- **String readLine()**

Do this

Download and install the textconsole jar file from this link:
https://dl.dropbox.com/u/2234170/courses/CSE1102/jars/textconsole-1.0.jar
Add a new variable in the build path for this jar file. Call it **TEXTCONSOLE**.

In the main method create an instance of **TextConsole** and store it in a variable (the class is in **jeff.textconsole**), but in order to do this you will need to call the static method **TextConsole.create()** instead of calling **new TextConsole()**. [*Nerd stuff, not on the exam*: The **TextConsole** instance must be created inside a concurrent thread, so the **TextConsole.create** method spawns a new thread, creates a new instance, and then returns the instance. You can look at the Java source code in the jar file if you like.]

Pass the text console instance as an argument to the **run** method. This means you must add yet another parameter to the **CommandInterpreter.run** method.

Inside the run method, replace every **System.out** with the name of the text console variable. For example, in my run method, the text console parameter variable is called **textConsole**, so I will change this statement:

```
System.out.println(agent.getName() + " is in " +
  agent.getLocation().toString());
```

to this:

```
textConsole.println(agent.getName() + " is in " +
  agent.getLocation().toString());
```

Also add a **TextConsole** parameter to the **Agent.usePortal** method, and change the **System.out** there also. Eclipse indicates that there is now an error in the **CommandInterpreter.run** method, but its suggestion how to correct the error should be the right one (a **TextConsole** instance must be supplied as an argument to the **usePortal** method).

Now with the text console you no longer need to use the **Scanner**. Get rid of the **Scanner** and change the call to **kbd.next()** (or however you called the **next** method) to be **textConsole.readLine()**.

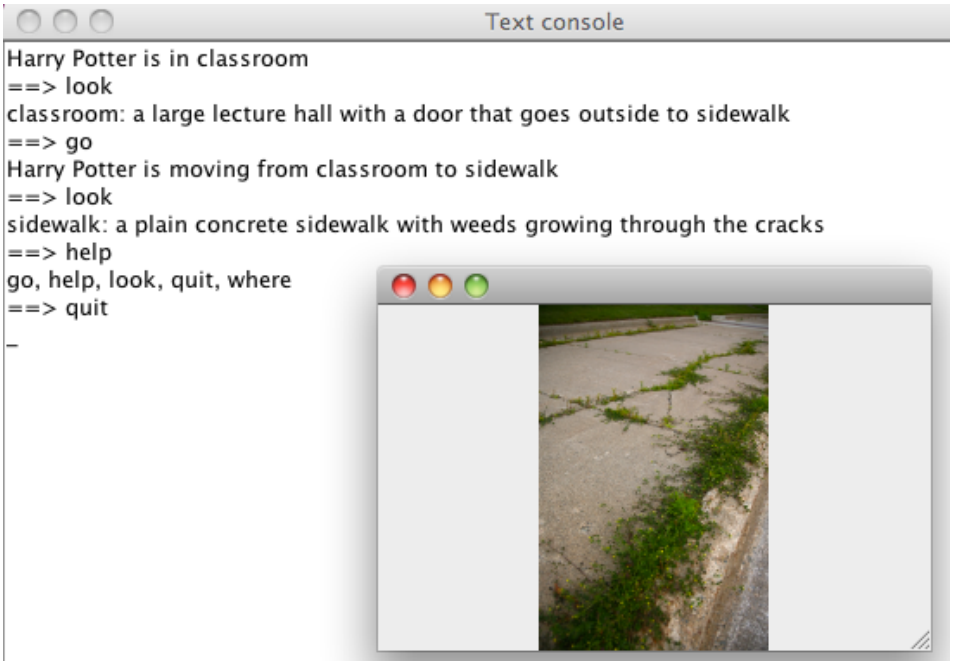
Finally, in the run method but below the end of the loop, add this statement:

```
System.exit(0);
```

Just by typing the quit command the run method's loop ends, but the text console does not end because it runs in a concurrent thread [beyond scope of this course]. The call to **System.exit(0)** ensures that the thread is terminated.

Testing

Test it. It should work.



6.9. Use the ini jar file

Summary

The ini jar file contains an **Ini** class that I have written for you that will load configuration data from a file and store it in memory. All the data is stored as strings. It is your responsibility to convert the configuration data into actual **Spaces**, **Portals**, and an **Agent**.

Do this

Yep, another jar file. This is the one I discussed in the background section of this document.
https://dl.dropbox.com/u/2234170/courses/CSE1102/jars/ini-1.0.jar
Add a new variable in the build path for this jar file. Call it **INIFILE**.

Create a class called **ConfigLoader**. Start with this definition:

```
public class ConfigLoader
{
  private Ini _ini;
  private HashMap<String, Space>  _spaces  = new HashMap<String, Space>();
  private HashMap<String, Portal> _portals = new HashMap<String, Portal>();
  private HashMap<String, Agent>  _agents  = new HashMap<String, Agent>();

  public ConfigLoader(File iniFile)
  {
    _ini = new Ini(iniFile);
  }
```

```
  public Agent buildAll()
  {
    //_buildSpaces();
    //_buildPortals();
    //_buildExits();
    //_buildDestinations();
    //_buildAgents();
    //return _selectStartAgent();
    return null;
  }
}
```

The **Ini** class is in **jeff.ini**.
The **HashMap** class is in **java.util**.
The **File** class is in **java.io**.

When a new **Ini** instance is created, it automatically loads the data from the disk file. Now all that is left is to read the data from the tables inside the **Ini** instance in the **_ini** variable, and create instances of things from them.

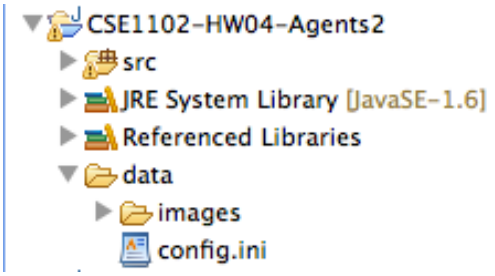
The load method refers to 6 undefined private methods. You will define these methods.

Testing
Make sure everything still works as before.

6.10. Create config.ini file in data folder
Summary
Before the **Ini** class can do its thing, there needs to be an **ini** file for it to read.

Do this
Create a new text file in Eclipse called **config.ini** and save it in the **data** folder in this project. This file must contain valid configuration data in the INI format. It's probably best to start with the example I gave in the background section (just copy & paste). After you get your program working, then you can change the ini file to represent your own set of spaces and portals.

Testing
Make sure that the **config.ini** file appears in the **data** folder:



6.10. Add _buildSpaces method

Summary

This method will create a number of **Space** instances based on the information that was loaded from the config.ini file. The **HashMap**s in the **ConfigLoader** instance are used to store **Space**, **Portal**, and **Agent** instances until the entire connected space can be built. A **HashMap** is like a dictionary: you give it a word, and it tells you the definition. Only here the word is the name of the instance (like "classroom"), so the **get** method called on the **HashMap** instance returns the **Space** instance that has that name.

Do this

This method is private, returns nothing, and has no parameters. All the information it needs is in the **Ini** instance, and it stores the **Spaces** it creates in the **HashMap**.

The **keys** method called on a **HashMap** will give you a collection of keys in a specific section. For example, in my config.ini file I have this **spaces** section:

```
[spaces]
classroom = a large lecture hall
sidewalk = a plain concrete sidewalk with weeds growing through the cracks
```

So this method call **_ini.keys("spaces")** will return the strings "classroom" and "sidewalk". Fortunately you don't have to worry about what form that collection is in (it's a **Set<String>** instance). All you need to do is iterate over that collection using a for loop:

```
for(String spaceName : _ini.keys("spaces"))
{
}
```

Whatever the keys (space names) are in the **spaces** section, this **for** loop will place one at a time in the **spaceName** variable and run the body of the loop.

In the body of the loop do this:

1. Get the description of the space: The descriptions are stored in the **spaces** section of

the INI file. Since the name of the space is already in the **spaceName** variable, the description is the value returned by **_ini.get("spaces", spaceName)**.

2. Get the image name of the space. That information is in the **images** section of the INI file.
3. Create a new **Space** instance using all the information you know: name, description, image file name. Use the value **null** as the space's portal. You don't have that information yet.
4. Place the space instance in the **_spaces** table for temporary safe keeping (by table I mean **HashMap** instance). Use this statement:
   _spaces.put(spaceName, spaceInstance);
   Remember: the **Ini** instance stores only strings. The **HashMap**s store instances of other classes.

Un-comment the call to **_buildSpaces()** in the **buildAll** method and place this statement below it:

```
System.out.println(_spaces);
```

Modify the main method
Delete all the statements that create new instances of **Space**, **Portal**, and **Agent**. Actually, here's what your **main** method should look like:

```
TextConsole  console     = TextConsole.create();
ImageWindow  imageWindow = new ImageWindow();
File         configFile  = new File("data", "config.ini");
ConfigLoader cl          = new ConfigLoader(configFile);
Agent        a           = cl.buildAll();
CommandInterpreter.run(console, imageWindow, a);
```

If you copy & paste that, be sure to put a comment there that says "given to us by Jeff" so that your TA understands that you're not trying to pass it off as your own work.

Testing

Run it. Here's the output in the Eclipse console window.

```
{classroom=classroom, sidewalk=sidewalk}
Exception in thread "main" java.lang.NullPointerException
      at CommandInterpreter.run(CommandInterpreter.java:13)
      at Main.main(Main.java:18)
```

The error is because we didn't build the Agent yet (and your line numbers will probably be

different), but the part before that is the contents of the **HashMap**. The keys are **classroom** and **sidewalk**, and the values are… also **classroom** and **sidewalk**?

Actually this is correct. The values are instances of **Space**, and when the **HashMap** displays the values it calls the **toString** method on the instance. The **toString** method in the **Space** class just displays the name of the space. It's sort of like this:

```
{classroom=classroomSpace.toString(), sidewalk=sidewalkSpace.toString()}
```

Delete the **println** statement after you verify that the method works.

6.11. Add _buildPortals method

Summary

This method will build **Portal** instances from the data in the **Ini** instance.

```
[portals]
door = outside
```

Do this

Similar to the **_buildSpaces** method:
For each portal name in the **portals** section:

1. Get the portal's description
2. Create a new **Portal** instance using the name and description. Use **null** for the portal's destination.
3. Store the instance in the **_portals** table using the portals name as the key.

Un-comment the call to **_buildPortals** in the **buildAll** method, and below that display the **_portals** table just for testing.

Testing

Here's mine. There's only one portal defined in my config.ini file:

```
{door=door that goes outside}
Exception in thread "main" java.lang.NullPointerException
      at CommandInterpreter.run(CommandInterpreter.java:13)
      at Main.main(Main.java:18)
```

6.12. Add _buildExits method

Summary

This method does not create any new instances. The Spaces and Portals are all stored in the tables. Recall how the exits are specified in the INI file:

```
[exits]
classroom = door
```

The key is the name of the space and the value is the name of the portal. All that needs to be done is to fetch both from the tables and assign the portal as the space's exit portal.

Do this

For each space name in the **exits** section:

1. Get the name of that space's exit portal from the **_ini** instance.
2. Get the actual **Space** instance from the table: **_spaces.get(spaceName)**
3. Get the **Portal** instance from the **_portals** table. Use the portal name, not the space name.
4. Set the space's portal to that portal instance.

Un-comment the call to **_buildExits** in the **buildAll** method.

Testing

This can't easily be tested. The portal's destination space is still **null**, and it causes the program to crash if you try to display it. Just proceed to the next step.

6.13. Add _buildDestinations method

Summary

This method works almost the same as the **_buildExits** method. It does not create any new instances. Recall how the destinations are specified in the INI file:

```
[destinations]
door = sidewalk
```

The key is the name of the portal and the value is the name of the space. All that needs to be done is to fetch both from the tables and assign the space as the portal's destination.

Do this

For each portal name in the **destinations** section: get the **Portal** instance that has the specified name, then get the **Space** instance that has the name of that portal's destination (hint: the portal's destination is currently **null**: where do you find that information?), then set that portal's destination to be that space.

If the space that get from the portal is equal to **null**, then display some kind of error message and then call **System.exit(1)**. This will terminate the program, because it can't proceed if it determines that the space is **null**.

Un-comment the call to **_buildDestinations** in the **buildAll** method.

Testing

Now that all spaces and portals are connected to each other you can check to see if the classroom space has been set up correctly. Add this statement below the **_buildDestinations()** method call:

```
  System.out.println(_spaces.get("classroom").toStringLong());
```

Here's what mine shows:

```
classroom: a large lecture hall with a door that goes outside to sidewalk
Exception in thread "main" java.lang.NullPointerException
      at CommandInterpreter.run(CommandInterpreter.java:13)
      at Main.main(Main.java:18)
```

6.14. Add _buildAgents method

Summary

This shows that there is one agent named **Harry Potter** and the name of the start location for that agent is **classroom**:

```
[agents]
Harry Potter = classroom
```

Do this

For each agent in the agents section: get the agent's starting location name, get the **Space** instance that has that name. If the space is **null**, show an error message and exit. Otherwise create a new **Agent** instance having that name and that start location. Store the agent in the **_agents** table. Use the agent's name as the key.

Testing

You can use **System.out.println(_agents)** in the **buildAll** method to see the contents of the table:

```
{Harry Potter=Harry Potter}
Exception in thread "main" java.lang.NullPointerException
      at CommandInterpreter.run(CommandInterpreter.java:13)
      at Main.main(Main.java:18)
```

Same as before: the **Agent.toString** method shows only the **Agent** instance's name, so that's what you see on the right side of the equal sign.

6.15. Add _selectStartAgent method

Summary

The config.ini file has one more section. It specifies what agent will be considered the central agent of the simulation.

```
[start]
agent = Harry Potter
```

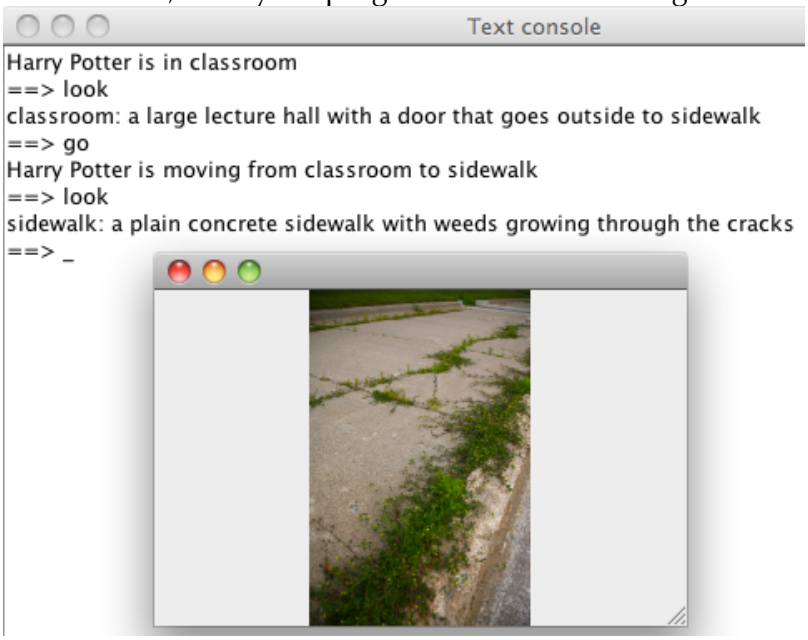
Do this

Fetch the name of the start agent from the **_ini** instance. Locate the agent having that name name. If the agent instance is null, display an error message and exit the program. Otherwise, return that agent from this method.

Un-comment the **return _selectStartAgent()** method call and delete the **return null** statement.

Testing

If this works, then your program will start working immediately.



Make sure you delete all the **System.out.println** methods in the **buildAll** method.

6.16. Create your own config.ini file

Summary

Now is your opportunity to get creative. You will change the config.ini file to be something of your own design, and you will also supply some images that go along with the spaces you create.

Either use the spaces and portals like you had in the last project, or create something new.

Do this

Use the interwebs to locate some images that are representative of the spaces in your project (only the spaces, not the portals or the agent). Please do your best to ensure that the images are not illegal to use (which is unfortunately, like, most images online).

An ideal alternative is to show off your mad MS Paint skillz. That means draw your own images. Actually I don't recommend MS Paint. Download GIMP or Inkscape some other drawing program. If you choose to draw your own images, they don't have to be wondrous works of art, but do put more than a few minutes into it. Creativity is more important than skill.

You could also draw some images on paper and then take pictures of the drawings. I believe you can buy crayons at the Co-op. Seriously, crayon drawings would be *awesome*.

For example, some of the images shown here are quite compelling, and probably didn't take too long to make.

Save the images as PNG or JPG files in the **data/images** folder. You can drag & drop the image files right into Eclipse, then choose the option to copy the images.

Testing

Make sure it all works.

7. Report

Create a Microsoft Word or OpenOffice Word file (or some other format that your TA agrees on -- ask him or her if you are not sure). Save the file with a .doc or .docx format.

At the beginning of the document include this information:

Project title goes here
CSE1102 Project *project-number*, *semester*
Your name goes here

The current date goes here
TA: *Your TA's name goes here*
Section: *Your section number goes here*
Instructor: Jeffrey A. Meunier

Be sure to replace the parts that are underlined above.

Now create the following three sections in your document.

1. **Introduction**
   In this section copy & paste the text from the introduction section of this assignment. (It's not plagiarism if you have permission to copy something. I give you permission.)
2. **Space diagram**
   Draw a sketch of your spaces and portals similar to the one you did last time, and include that sketch here. Use the same diagram if your spaces and portals are the same as last time.
3. **Space images**
   Copy & paste each of your space images here. Resize the images so they don't take up a lot of room; not more than 1/4 page each. Label each image with the name of the space.
4. **Source code**
   Copy & paste the contents of your Java file(s) here. You do not need to write anything.

8. Submission

Submit the following things things on HuskyCT:

1. Your exported Eclipse project: make sure that the **data** folder is included when you export the project.
2. The report document.

If for some reason you are not able to submit your files on HuskyCT, email your TA before the deadline. Attach your files to the email.

9. Notes

Here are some notes about this project:

- The **Build path** variables in Eclipse must be named exactly as I specify, otherwise the program will not run after your TA imports your project. You do *not* want to make extra work for your TA.

10. Grading Rubric

Your TA will grade your assignment based on these criteria:

- (2 points) The comment blocks at the beginning of the class files are correct.
- (11 points) The program is written correctly and works correctly (i.e., it doesn't work correctly by accident).
- (4 points) The program is formatted neatly.
- (3 points) The document contains all the correct information and is formatted neatly.

11. Getting help

Start your project early, because you will probably not be able to get help in the last few hours before the project is due.

- If you need help, send e-mail to your TA immediately.
- Go to the office hours for (preferably) your TA or any other TA. I suggest you seeing your own TA first because your TA will know you better. However, don't let that stop you from seeing any TA for help.
- Send e-mail to Jeff.
- Go to Jeff's office hours.