Comp 401 - Assignment 12: Wait Notify, Generics, Etc. (Last One!)

Date Assigned: Sat Nov 14, 2015

Completion Date: Wed Dec 2, 2015 (Last day for Submitting All Work)

Early Completion Date: Mon Nov 30, 2015 (Monday)

This assignment will change as we cover new material. The required part addresses wait and notify and generics. There is extra credit for exceptions, recursive descent, undo. I will probably add extra credit for factories soon.

Animation and Threads: Wait and Notify	PowerPoint PDF YouTube	Commands Chapter	Visualization	lectures.animation.threads.wait_notify Package
Generics, Adapters, Basic Delegation	PowerPoint PDF YouTube	Generics chapter	-	lectures.generics Package
Exceptions	PowerPoint PDF YouTube	Exceptions Chapter	-	-
Factories	PowerPoint PDF -	-	-	-

lectures.exceptions Package

Coordinating Animation Starts

This part of the assignment will allow you to coordinate the start of the animations of the avatars.

Add four additional methods to the interpreter to animate Arthur, Galahad, Lancelot, and Robin in separate threads. Let us call these methods as the waiting animation methods. Tag them as "waitingArthur," "waitingGalahad," "waitingLancelot" and "waitingRobin" respectively. These are like the asynchronous animation methods you have previously created.

The only difference is that these methods ensure that before the animation loop is executed, the waitForProceed() method of an instance of BroadcastingClearanceManager is called. The waitForProceed() call is not made directly by the waiting methods, as they create threads that perform the animations. It is made by the command objects executed by the created threads The same instance of BroadcastingClearanceManager must be used for all of these waits. The main method will create this clearance manager and pass it to the command interpreter as a constructor argument, which will pass it to the command object.

You should not create a new version of the "AnimatingCommand" command class. You should extend your previous class to take an additional constructor parameter that decides if waiting should be done.

Now add yet another method to the command interpreter that calls the proceedAll() method of the BroadcastingClearanceManager. Tag this method as "startAnimation".

These five methods will allow you to coordinate the start of the animations of the avatars. You will first execute the four waiting animation methods, which will wait for the animations start method to execute proceedAll() before performing the animations. Thus, when you invoke the start animation method, all four animations will start simultaneously.

ObjectEditor comes with a clearance manager. Do not use it. Instead use the one described in class and make sure the source code for it is in your project so you can trace its actions in case of problems. Make a copy of the necessary JavaTeaching code, please do not reference the project directly, as that will make our grading easier. Do not change the name or methods of BroadcastingClearanceManager so we can easily find and test it.

Generics

Use generics to define your table class and elaborate this class with different type parameters to (a) map avatar names to avatars and (b) implement the extended grammar with define and call commands (if you do the extended grammar). Tag your table class as "generic".

As the discussion above implies, you must modify the table class you implemented in a previous assignment. You cannot use a Map. You should not re-implement a table in terms of a Map.

Deleted: either

Deleted: or the animation methods in the animator objects, which are called by the run methods of the command objects. In either case, t

Deleted:

Deleted:

Deleted: or animator

Deleted: do not have to

Deleted: s

Deleted: the animator or

Deleted: es

Deleted: can

Deleted:

Deleted: es

Deleted: the

Deleted: a

Deleted: method

Deleted: (if you do the extended grammar)

Extra Credit: Lockstep Animation Methods

In this part of the assignment, the guard will coordinate the animations of the knights in the manner illustrated in my demo. It will provide the beats to which the other avatars do their movements.

You will create four new animation methods in your command interpreter to animate the four knights. Let us call these the lockstep animation methods. Their tags are "lockstepArthur," "lockstepGalahad," "lockstepLancelot" and "lockstepRobin" respectively.

These are like their asynchronous counterparts except that instead of calling sleep, their execution result in calls (in the animation objects) to waitForProceed() on the global clearance manager.

There are two ways to implement the lockstep part,. One is to take the existing command and animation objects to get take an extra parameter that specifies if a sleep or waitForProceed, or proceedAll should be called. Take the second approach to gain more experience with inheritance. Ideally an abstract class should be defined whose abstract methods decide on sleep or wait. Tag the command object and animator for these method as "CoordinatedAnimationCommand" and "CoordinatedAnimator" respectively.

Add to your command interpreter a variation of the clapping guard extra credit of the previous assignment. Tag this a method tagged "lockstepGuard" method– the only difference is that after each sleep() call, proceedAll() will be called in the global clearance manager.

Tag the command object for this method as "CoordinatingAnimatingCommand and the animator as "CoordinatingAnimator."

Both animators should define a method tagged "animateAvatar".

Extra Credit: More Recursive Descent Parsing (Changed a bit from Assignment 11)

Once you have the steps above working, attempt this part, which shows you a new use of the table class. Here, you will create additional command objects and parse a more sophisticated grammar. You will not learn new kind of material here, but it should be satisfying to do this part to create an interesting project. As it conceptually belongs in parsing, I am putting it here. However, it will be graded with assignment 12 to give you more time to implement it.

The following are the additional command objects:

RotateLeft(Right)Arm Atomic Commands: Tags: "RotateLeftArmCommand" and "RotateRightArmCommand". The constructor takes a parameter that specifies an avatar object and an int parameter (in that order) that specifies by how much the left(right) arm of the avatar should be rotated. The run method the class rotates the left (right) arm by the specified amount.

Deleted: y will

Deleted: call

Deleted: the

Deleted: . This method will also be like its asynchronous counterpartbe like the lockstep methods above

Deleted: it will call

Deleted: To implement this method, define a new command class, tagged

Deleted: ", which is like "AnimatingCommand", except that it calls proceedAll() rather than waitForProceed

Deleted: ¶
-9}¶

Sleep Atomic Command: Tag :"SleepCommand". This is a command class that takes in its constructor a long value representing the sleep time. The run method of the class simply calls ThreadSupport.sleep() to sleep for sleep time. If you use this command for animation, be sure to execute it as a part of the thread command, otherwise the AWT thread will not paint the result of commands executed before (and after) the sleep until the complete command entered by the user has been executed.

Define Composite Command: Tag: "DefineCommand" This command associates a command with a name, which can be used by the call and thread commands. The constructor of this class takes three arguments, a String, a command, and a table, which we will refer to as a command name, command body, and environment, respectively. The run method of the class simply calls the put method in the environment to associate the command name with the command body.

Call Composite Command: Tag: "CallCommand". The constructor of this class takes two arguments, a String and a table, which are a command name and environment, respectively. The run method of this class calls the get() method in the environment to find the command body associated with the command name, and calls the run method of the command body.

Thread Composite Command: Tag: "ThreadCommand". The constructor of this class takes two arguments, a String and a table, which are a command name and environment, respectively. The run method of this class calls the get() method in the environment to find the command body associated with the command name, and creates and starts new thread that executes the command body (asynchronously). In other words, the run command creates a new Thread instance, passing to the Thread constructor the command body, and starts the thread. As mentioned above, you will need this command to prevent the AWT thread from blocking. As your command objects all implement the Runnable interface, you can simply pass the command object representing the body to the constructor of the Thread class.

ProceedAll Command: Tag: "ProceedAllCommand". The constructor takes the broadcasting clearance manager as an argument, and executes the proceedAll method on the manager.

Formatted: Font: Italic

Use these command objects to do recursive descent parsing of the following extensions to the grammar given above:

1. <Command> → <Rotate Left Arm Command> | <Rotate Right Arm Command> | <Sleep Command> | <Wait Command> | <ProceedAll Command> | <Define Command> | <Call Command>| <Thread Command>
2. <Rotate Left Arm Command> → rotate-left-arm-token word-token number-token
3. < Rotate Right Arm Command> → rotate-right-arm-token word-token number-token
4. <Sleep Command> → sleep-token number-token
5. <Define Command> → define-token word-token <Command>
6. <Call Command> → call-token word-token
7. <Thread Command> → thread-token word-token
8. < ProceedAll Command > → proceedAll-token

Tag the parser methods using the conventions we have seen so far. Thus, the tags should be: "parseRotateLeftArm", "parseRotateRightArm", "parseSleep", "parseDefine", "parseCall", "parseThread".

Examples of commands following the extended syntax are:

```
define guardArmsIn {rotateLeftArm guard 9 rotateRightArm guard -9}
define guardArmsOut {rotateLeftArm guard -9 rotateRightArm guard 9}
define beat {call guardArmsIn proceedAll sleep 1000 call guardArmsOut sleep 1000
define beats repeat 10 call beat
thread beats
```

This sequence of command results in an animation in which the guard claps to a certain beat, and on each clap, notifies all threads waiting for clearance from the clearance manager.
The first two commands associate the command names guardArmsIn and guardArmsOut with two different command lists. The third command associates the command name, beat , with a command list in which calls to the earlier defined commands are made. The fourth command associates the command name, beats, with a repeat command that calls beat. The final command starts a new thread to execute the beats command. Your main class should demonstrate this sequence to get credit for this extra credit part.

The mapping from each-non terminal to the associated command object is straightforward. The only aspect that perhaps needs explanation is the <word-token> in commands 13-15, which is a command name. The command interpreter should create a single environment (table) for all of the commands objects it creates. Also, as mentioned above, it should receive the clearance manager as a constructor argument, which is given to it by main.

Extra Credit: Declaring, Throwing, and Catching Exceptions

Do this part if we cover exceptions in class or you are willing to read about them on your own. You can browse down my web page to look at my notes.

If you have not done so already as part of extra credit, define a special readonly dependent String property in the parser to show the scanning and/or parser errors encountered while processing the command string. The setting of the error string should be done in the setter for CommandText in the parser. Use exceptions to communicate errors between the methods that detect them and the CommandText setter.

When a scanner or parser method detects an error, it should not print the error or return a special value to its caller. Instead, it should throw either a scanning or parsing exception, depending on the kind of error, and set the message of the exception to indicate the error details. Thus, the method that catches an error will determine the error message, but not when or how the error message is displayed. This decision will be made by the CommandText setter.

Formatted: Indent: Left: 0.25", No bullets or numbering

Deleted:

Deleted: the command string

Deleted:

Deleted: command setter

Deleted: command

Tag the classes defining these exceptions as “ScanningException” and “ParsingException” respectively. They should be subclasses of IOException.

Extra Credit: Static Factory Methods

Define a static-factory class , tagged ‘SingletonsCreator” that defines parameterless static factory methods, tagged “scannerFactoryMethod”, “parserFactoryMethod”, “bridgeSceneFactoryMethod”, “avatarTableFactoryMethod,” “commandInterpreterFactoryMethod,” and “broadcastingClearanceManagerMethod”. Each of these methods should return an instance of the type denoted by its name. For example, the method tagged scanner factory method should return an instance of the ScannerBean class. The method should ensure that multiple instances of the associated type are not created. Thus, the scanner factory method should return the same instance each time. Also the instance should not be created until it is needed. A factory method can call another factory method. For instance, the factory method for creating the parser should call the factory method for creating the scanner.

Change the remaining code in your project to ensure that the scanner, parser, bridge scene, avatar table, command interpreter and broadcasting clearance manager are created through the factory methods rather than directly by instantiating the corresponding singleton classes. Many of the classes in your code expect instances of the classes instantiated by factory methods in constructor arguments. Define additional constructors in these classes that do not expect these instances as parameters and call the factory methods to get references to them. For example, the command interpreter has a constructor that expects a bridge scene as a parameter. Define another constructor in the command interpreter that does not expect the bridge scene as a parameter. Instead, it sets its bridge scene instance variable to the value returned by the associated factory method.

You should not change the visibility of the classes instantiated by the factory methods.

Extra Credit: ObjectEditor Factory Class and Instance Factory Method

For this part, you will have to download the latest version of oeall22.jar. Create a subclass of the ObjectEditor factory class, bus.uigen.widgets.swing.SwingTextFieldFactory. The class has a factory method with the signature:

```
protected JTextField createJTextField(String aText) ;
```

Override this method to return an instance of a JTextField displaying the String passed as an argument to this method. You can do so by calling:

new JTextField(aText);

Before you return the JTextField, set its Background and Foreground properties to colors of your choice that are not the default colors. Some common colors are defined as constants such as BLUE and GREEN in java.awt.Color. Feel free to change other aspects of the text field.

Tag this class as “CustomSwingTextFieldFactory”.

Deleted:

Formatted: Normal

Formatted: Normal

Formatted: Normal, Don't adjust space between Latin and Asian text, Don't adjust space between Asian text and numbers

If you do this part, make sure your main class calls ObjectEditor.initialize() and assigns an instance of this class to the TextFieldFactory static property of the factory selector, bus.uigen.widgets.TextFieldSelector. The factory should be assigned before any object is displayed by ObjectEditor. The result of doing should be that the command interpreter text field (and any othe text field generated by ObjectEditor) should have the colors chosen by you.

Deleted: ¶

Formatted: Normal, Don't adjust space between Latin and Asian text, Don't adjust space between Asian text and numbers

Extra Credit: Undo

Deleted: ¶

Do this part if we cover undo in class or you are willing to read about them on your own. Support the following extended grammar:

<Command> → <Undo Command> | <Redo Command>

<Undo Command> → undo-token

<Redo Command> → redo-token

An undo(redo)-token is the word "undo"/"redo". Tag the command classes for these commands using the conventions given earlier.

Allow the move command to be undone and redone by these commands. Tag the command classes for undo and redo for it as "UndoCommand" and "RedoCommand", respectively.

Deleted: class for it as "Undoable".

Constraints

You should have no errors or warnings from ObjectEditor at this point. If ObjectEditor generates what you think are wrong or unreasonable, download the newest version. If that also gives you what you think are spurious messages, post on Piazza and if you do not get a quick answer contact the TAs and/or me through help401 or office hours – preferably help401. *If the TAs cannot solve your problem, contact me*. If I cannot solve your problem, you will be excused the error/warning.

Animating Demoing Main Class

To demonstrate your work, write a main class that creates a scene object displays animations of it using the console scene view and the painting view, and displays the command interpreter user interface. Specifically, the main class:

1. Instantiates a scene object and displays it using your painting view object or ObjectEditor.
2. Creates an instance of the broadcasting clearance manager and displays it using ObjectEditor.
3. Instantiates a command interpreter object and displays it using ObjectEditor (and *not* your custom command interpreter user interface). *This is important from a grading point of view.* As mentioned before, the command interpreter constructor will be passed the instance of this clearance manager as an argument.

4. Demonstrate the waiting methods. Your program should start all four waiting methods and the TAs will press the button in the clearance manager to start the corresponding animations.
5. For extra credit demonstrates the lockstep methods. Do so by starting the lockstep methods for the avatars and then the lockstep method for the guard.
6. Assigns different extra credit commands to the editable property of the command interpreter to show what your interpreter can parse and process. After each setting of a new command, call waitForProceed() in the broadcasting clearance manager. The waitForProceed should be called by main not by the command interpreter setter This means that you or the TAs can click on the proceed button to see the effect of each command. The TAs will assign their own strings to the command interpreter. It is possible your interpreter will fail on some of those. Therefore it is very important for you to assign test cases to show what does work.
    a. For the extended grammar show some variation of the thread beats example above, where the rotation degrees of course should be different. This means you should assign some sequence of commands to the parser property that makes the Guard clap to some beat and execute proceedAll after each beat to make the rest of the avatars do lockstep movements. Before assigning the commands to the interpreter, call the lockstep animation methods in the command interpreter of all avatars but the guard. The result of executing all of commands will be to provide the beats for the four avatar animations. You will get all or no extra credit for the extended grammar based on whether you demo this sequence.
    b. For undo/redo and exceptions, assign command sequences that illustrate their working.

Submission Instructions

- These are the same is in the previous assignment. The TAs will run the main method to see the test cases animate.
- They will also get a reference to your command interpreter (through ObjectEditor.edit()) and set different strings to its editable property.
- Be sure to follow the conventions for the name and package of the main class.

Good luck with your last assignment!