# Comp 401 - Assignment 11: Mainly Parsing (Extra Credit)

**Early Completion Date: Wed, Nov 15, 2017 (+5%)**

**Completion Date: Fri, Nov 17, 2017**

This assignment covers three topics: abstract methods, recursive-descent parsing and synchronized methods. Recursive-descent parsing, in turn, requires your understanding of grammars and composite command objects, that is, command objects that refer to other command objects.

| | | | | | |
|---|---|---|---|---|---|
| Inheritance: Abstract Classes | PowerPoint<br>PDF<br>YouTube | Inheritance Chapter | | Mainly Parsing<br>Checks File | lectures.inheritance.abstract_classes Package<br>Git (Abstract Classes) |
| Parsing and Grammars | PowerPoint<br>PDF<br>YouTube | More Inheritance Chapter | | - | lectures.parsing_grammars Package<br>Git (Parsing and Grammars) |
| Animation and Threads: Synchronized Methods | PowerPoint<br>PDF<br>YouTube | Commands Chapter | Visualization | - | lectures.animation.threads.synchronized_methods Package<br>Git (Synchronized Threads) |

## Abstract Classes

In your token and shape inheritance hierarchies, declare classes that are not to be instantiated directly as abstract classes.

## Approach, Pass, Fail Command Objects

Create three new command classes, tagged "ApproachCommand," "PassCommand," and "FailCommand," respectively that represent three new commands your parser will understand. As before these classes will implement the Runnable interface. The approach command will define a constructor that specifies a scene object and an avatar object, in that order, and the other two commands will define constructors that take the scene object as a parameter. As before, the run methods of these command classes will execute the associated scene method, using any values passed in the constructors.

## "Command List" Composite Command

Create a class, tagged "CommandList," that stores a dynamic list of command objects, which in this project are instances of the Java Runnable interface. You should use an ArrayList to store the elements of the list, but make sure to declare it as a List. The class should provide a method, tagged "add," to append a new (Runnable) command to the list, passed as an argument to add. It should have constructor that takes no arguments, which of course can be the default constructor.

The class should not only store instances of Runnable but also implement Runnable (which means it can implement some subtype of Runnable). The run method of this class should simply invoke the run method of each element of the list in order. Thus, if a command list consists of an approach command followed by a say command, then the run method of the command list will first invoke the run method of the approach command and then the run method of the say command.

Since a command list is a command consisting of commands, it is an example of a *composite command*. In contrast, the say and move commands are *atomic commands*, as they do not contain other commands. As a command list consists of arbitrary commands, it can contain not only atomic commands such as the move and say commands, but also composite commands such as a command list and other composite commands mentioned below. This means that the add method of the command list should take a Runnable as its argument.

## Repeat Composite Command

Create another composite command class, tagged "RepeatCommand", whose constructor takes two arguments: an int and a command object (that is, a Runnable implementation), representing a count and repeatable command, respectively. The run method of this composite command executes the repeatable command count number of times. Thus, if the two arguments of the constructor are 5 and a move command object, then the run method executes the move command 5 times.

## Basic Recursive Descent Parsing

You should change your command interpreter to do recursive descent parsing of the following grammar, which is an extension of the one you have supported so far:

1. <Command> → <Move Command> | <Say Command>
   | <Approach Command> | <Pass Command> | <Fail Command>
   | <Command List> | <Repeat Command>
2. <Say Command> → say-token quoted-string-token
3. <Move Command> → move-token word-token number-token number-token
4. <Approach Command> → approach-token word-token
5. <Pass Command> → pass-token
6. <Fail Command> → fail-token
7. <Command List> → start-token <Command>* end-token
8. <Repeat Command> → repeat-token number-token <Command>

Recall that the start and end tokens are '{' and '}'.

Thus, as before the following commands are legal:

move Arthur 2 3

say "Quest?"

In addition, the following commands are legal:

approach Arthur
passed
failed
{ move Arthur 2 3 approach Arthur say "Name?" }
{ move Arthur 2 3 { approach Arthur } say "Name?" }
repeat 5 move Arthur 2 3
repeat 5 { move Arthur 2 3 move Galahad 5 6 }
repeat 4 repeat 5 { move Arthur 2 3 move Galahad 5 6 }

Your command interpreter will not directly do the parsing. Instead, it will use an instance of a new class, tagged, "Parser" that performs this task. This means you must move your parsing code to this class.

Implement a static method, tagged "parserFactoryMethod", in 'SingletonsCreator" that can be used to create an instance of the parser, which is used by the command interpreter to get a reference to it.

This is the most difficult part of the assignment, so please do it thoughtfully, and take time to understand the material on recursive descent parsing. As you will be doing recursive descent parsing, you will define a separate parsing (instance) method for each of the non-terminals given above. The non- terminals here are: <Command>, <Say

Command>, <Move Command>, <Approach Command>, <Pass Command>, <Fail Command>, <Command list>, and <Repeat Command>. The methods that parse them should be called "parseCommand,", "parseSayCommand," "parseMoveCommand,", "parseApproachCommand," "parsePassCommand", "parseFailCommand," "parseCommandList", and "parseRepeatCommand", respectively.

The parse methods for <Say Command>, <Move Command>, <Command list>, and <Repeat Command> will return the Say, Move, CommandList, and RepeatCommand object, respectively. However, the return type of these methods will be the most specific interface of these objects – (Runnable or CommandList) to follow our rule that interfaces rather than classes should be used as types. The parse method for <Command> is simply a dispatching method, calling one of the other seven parse methods to determine the return value, based on the next token. Thus, it will return whatever Runnable instance the called method returns.

Some of these methods will be mutually recursive. The parse method for <Command> will look at the next token and call one of the other parse methods based on this token. Conversely, the parse methods for <Command List> and <Repeat Command> will call the parse method for <Command> to parse the component commands.

The parser class should defined an editable String property called "CommandText" that, when set, does the parsing after using the scanner to scan the string. The result of the parsing should be returned by the getter of the readonly property, CommandObject. Any error reports (extra credit part from a previous assignment) should be stored in a readonly property called Errors. Thus, the command interpreter no longer interacts with the scanning code – this task is done by the parser, which instantiates the scanner. The parser also instantiates and fills the table mapping avatar names to objects. The setter for the Command property of the command interpreter sets the CommandText property of the parser and calls the getter for the CommandObject property to process or interpret the command. The command interpreter uses the Errors property of the parser to set its own error property (extra credit).

You can assume the user makes no errors.

## Iterator-like Methods

You will have to make sure that each of the parsing methods knows the index of the next token to look at in the array of tokens received from the scanner. This index, together with the array, should be global variables. Define the following methods to manipulate these variables:

1. next(): this method returns the token at the current index position, and increments the global index;

2. peek(): It has the same behavior as next() except that it does not change the global index. Thus, it can be used to look at the next token without "consuming" it. This method should be called by next() so that the two methods share code.
3. hasNext(): returns true if there is a next token, that is, if the next call to next() (or peek()) will not give an index-out- of-bounds. This method should not change any global variable.
4. reset(): sets the index so the next call to next() will return the first token in the array.

None of these methods takes an argument. These methods are inspired but not identical to the ones provided by the Java Iterator interface. These are the only methods in the parser that should access the global array and index.

## Synchronized Animations of Same Avatar

In the previous assignment, you added four parameterless methods to the command interpreter to animate Arthur, Galahad, Lancelot, and Robin in separate threads. We referred to these methods as the asynchronous animation methods. When you use these methods, it is possible for you to start two animations concurrently that manipulate the same avatar. In this part of the assignment, prevent this from happening. Thus, it should be possible for Arthur and Galahad to be animated at the same time, but not for Arthur to be manipulated concurrently by two different animation threads. This means that you must now create a unique animator instance for each avatar, which is shared by all command objects (created by the command interpreter) that animate the avatar. Use the Java **synchronized** keyword in the animator to prevent synchronized animations of an avatar from happening concurrently. The tags of the methods remain the same.

## Extra Credit Parsing: Signed Numbers

If you supported the following rule in the command interpreter assignment:

<Number> → number-token | +token number-token | -token number-token

then replace the number-token in the grammar above with <Number>.

To do recursive descent parsing for <Number>, make the parse method for it return an int value (rather than a command object). Tag the parser method for this non-terminal as "parseNumber".

Example of use of this feature:

{ move guard + 10 - 100 }

The second command above assumes you have done the extra credit mentioned below.

## Constraints

Same as in earlier assignments. As mentioned earlier, the global array and index should be accessed only by the iterator-inspired methods.

## Hints

*Be sure to reset the index each time a new string is parsed and a new array returned from the scanner bean.* The peek() method will be useful for parsing command lists. It can be also used in parseCommand(). The hasNext() method will be useful for error detection.

## 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. Creates (using the factory method) a scene object and displays it using your painting view object or ObjectEditor.
2. Creates 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.*
3. Demonstrates the new versions of the asynchronous methods. You should make two calls to (a) different asynchronous animation methods (animating different avatars) and (b) the same asynchronous method (animating the same avatar). Thus, you should make three calls that, for instance, animate Galahad once and Lancelot twice. The result should be that two animations of different avatars should execute concurrently (e.g. Arthur and Lancelot) and after one of these completes (e.g. Lancelot), it should immediately start again.
4. Assigns different commands to the editable property of the command interpreter to show what your interpreter can parse and process. Make your command interpreter an observable if it is not already so, so that without calling ObjectEditor.refresh(), its properties are updated when its editable property is assigned by the main method. You will not be able to see individual changes made by each command, as you we not yet have a sleep command:
    a. For the basic grammar, you must show a command list that has a repeat command that has a command list.
    b. For signed numbers, show an example of signed numbers in the basic grammar test.

## 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.