Creating a New PhyloNet Command

This tutorial will guide you through adding a new example command to PhyloNet that can be executed from within the `PHYLONET` block of a NEXUS file.

Step 1. Prepare Existing Source

The first step in authoring a new command is to download the PhyloNet source code and its binary dependencies. The most recent version of the tool and its source can be found here.

PhyloNet also has a few binary dependencies: anltr-runtime, mockito, and apache-commons-io.

After downloading the PhyloNet source files and dependency libraries make sure you can compile PhyloNet. JDK 1.6 or later is required.

Step 2. Create Your Command Class and extend `CommandBase`

In our example we will create a new command `CountNodes` that computes the number of nodes in a given network.

The first step in this process is to create our `CountNodes` class and have it extend `edu.rice.cs.bioinfo.programs.phylonet.commands.CommandBase`:

```
package com.example;

import edu.rice.cs.bioinfo.programs.phylonet.commands.*;
import edu.rice.cs.bioinfo.library.language.pyson._1_0.ir.blockcontents.*;
import edu.rice.cs.bioinfo.library.language.richnewick._1_0.ast.*;
import edu.rice.cs.bioinfo.library.programming.*;
import java.io.*;
import java.util.*;

public class CountNodes extends CommandBase
{
   public CountNodes(SyntaxCommand motivatingCommand, ArrayList<Parameter> params,
                     Map<String,NetworkNonEmpty>  sourceIdentToNetwork, Proc3<String, Integer, Integer>
errorDetected)
    {
      super(motivatingCommand, params, sourceIdentToNetwork, errorDetected);
    }
    ...
}
```

The new command class should provide the illustrated public four argument constructor with corresponding call to `super`. When a command is encountered in a `PhyloNet` block within a NEXUS file, it is this constructor that will be called to create the command instance.

2.1 Implement Min and Max Params

By inheriting from `CommandBase` your class will have to implement the following methods:

```
protected int getMinNumParams()
{
    ...
}

protected int getMaxNumParams()
{
   ...
}
```

which define the minimum and maximum number of parameters allowed for your command in the NEXUS file. It is important to keep in mind that most commands in PhyloNet allow an optional final parameter for redirecting command output to a file instead of the console. So generally the maximum number of allowable parameters is one more than the number supported by the command itself. Our example `CountNodes` command will take a minimum of one parameter (the name of the network to scan) and at most two parameters (the second being the optional file destination of our command's output).

```
protected int getMinNumParams()
{
  return 1;
}

protected int getMaxNumParams()
{
  return 2;
}
```

2.2 Implement `checkParamsForCommand()`

By inheriting from `CommandBase` your class will have to implement `checkParamsForCommand()` which provides an opportunity to perform context sensitive analysis over the command's parameters prior to execution. The function should return `true` if no errors are detected and `false` if errors are detected. It should also report any discovered errors to the inherited `errorDetected` member.

For our command, we need only to check only that the given network identifier is in fact defined in the NEXUS document:

```
protected boolean checkParamsForCommand()
{
   int expectedNetworkNameParameterIndex = 0; // network ident should be first parameter in command.

   // automatically checks for network existance and reports any errors to this.errorDetected.
   NetworkNonEmpty network = this.assertAndGetNetwork(expectedNetworkNameParameterIndex);

   if(network == null) // user specified network name is invalid
   {
       return false;
   }
  _networkToScan = network; // added field to class to retain network to scan.
  return true;

}
```

Note that we have added a field to our class `_networkToScan` to retain a reference to the specified network to scan. This will be useful in our next step.

2.3 Implement `executeCommandHelp`.

The actual execution of the command takes place within the `executeCommandHelp` method. In our example we will put the command execution code in this method directly; however, it is advisable not to implement the core logic of larger commands here but instead to delegate that computation to a helper class. The single `displayResult` parameter is a function for displaying text results of the command to the user. The first call to `displayResult` should begin with a newline character to preserve the tool's output formatting. These results will be automatically displayed on the console or sent to an output file based on the user's preference.

```
protected void executeCommandHelp(Proc<String> displayResult) throws IOException
{
    String eNewickNetwork = NetworkTransformer.toENewick(_networkToScan); // convert NEXUS network to extended
newick string

    /*
     * convert extended network string to Network
     */
    edu.rice.cs.bioinfo.programs.phylonet.structs.network.io.ExNewickReader<String> enr =
      new edu.rice.cs.bioinfo.programs.phylonet.structs.network.io.ExNewickReader<String>(
          newStringReader(eNewickNetwork));

    edu.rice.cs.bioinfo.programs.phylonet.structs.network.Network<String> net;
    try
    {
      net = enr.readNetwork();
    }
    catch(Exception e)
    {
      throw new RuntimeException(e);
    }

    /*
     * count and display number of nodes in network
     */
    Iterable<edu.rice.cs.bioinfo.programs.phylonet.structs.network.NetNode<String>> allNodes = net.dfs();
    int nodeCount = 0;
    for(Object node : net.dfs())
    {
      nodeCount++;
    }

    displayResult.execute("\nNetwork contains " + nodeCount + " nodes.");
}
```

Step 3. Update `CommandFactory`

The next step is to update `edu.rice.cs.bioinfo.programs.phylonet.commands.CommandFactory` to construct instances of the `CountNodes` command. When examining the existing structure of the `CommandFactory` you will discover an `if`/`else if` chain within the `make` method:

```
public class CommandFactory {



    public static Command make(SyntaxCommand directive, Map<String,NetworkNonEmpty> sourceIdentToNetwork,
                               Proc3<String, Integer, Integer> errorDetected, Random rand)
    {
        ...

        if(lowerCommandName.equals("symmetricdifference") || lowerCommandName.equals("rf"))
        {
            return new SymmetricDifference(directive, params, sourceIdentToNetwork, errorDetected);
        }
        else if(lowerCommandName.equals("lca"))
        {
            return new LCA(directive, params, sourceIdentToNetwork, errorDetected);
        }
        ...
    }
```

To add a new command to the factory, select a lower case string name to represent the command in a NEXUS file and append an entry of the command in the chain. In our example we will name the `CountNodes` command simply `"CountNodes"`:

```
public class CommandFactory {



    public static Command make(SyntaxCommand directive, Map<String,NetworkNonEmpty> sourceIdentToNetwork,
                               Proc3<String, Integer, Integer> errorDetected, Random rand)
    {
        ...
        else if(lowerCommandName.equals("nexus_out"))
        {
            return new NexusOut(directive, params, sourceIdentToNetwork, errorDetected);
        }
        else if(lowerCommandName.equals("countnodes"))
        {
           return new CountNodes(directive, params, sourceIdentToNetwork, errorDetected);
        }
        ...
```

This will instruct PhyloNet to create instances of the new command when named within a `PHYLONET` block.

Step 4. Compile and Test

At this point inclusion of a new command is complete. After recompiling PhyloNet the new command should be available. For example, the following NEXUS file will now be processed by PhyloNet:

```
#NEXUS

BEGIN NETWORKS;

Network net = ((a,(b,(c)x#1)M)N,((x#1,d)J,e)Z)R;

END;


BEGIN PHYLONET;

CountNodes net;

END;
```