Each node represents a person, an individual and the edges represent relationships between those people.

Each node has a number associate with it, N1 through N5, and each edge has a corresponding number associated with it, E1 through E5.

Edges are relationships such as Harry is known by Tom, or Julian is coworker of Harry.

What we want is a script that we can process with Neo4j in order to create an actual graph network.

Step 2: Introduce equivalent text representations. The graph we saw in the previously examined, can be simply represented as following:

==========

Five Nodes

N1 = Tom

N2 = Harry

N3 = Julian

N4 = Michele

N5 = Josephine

Five Edges

e1 = Harry 'is known by' Tom

e2 = Julian 'is co-worker of' Harry

e3 = Michele 'is wife of' Harry

e4 = Josephine 'is wife of' Tom

e5 = Josephine 'is friend of' Michele

==========

Step 3. Build on text representations. After making a simple representation of our network, now we can proceed with a notation structure to describe the five edge relationships.

==========

Text representation

N1 - e1 -> N2

N2 - e2 -> N3

N2 - e3 -> N4

N1 - e4 -> N5

N4 - e5 -> N5

==========

You can see for example that the first relationship is clearly representing that Tom (N1) is connected through an edge with Harry (N2) because Harry is known by Tom (e1). The same logic goes for the rest of the relationships.

Now we can take another step into a more technical description:

==========

Technical pseudo-code

N1:ToyNode - e1 -> N2:ToyNode

N2 - e2 -> N3:ToyNode

N2 - e3 -> N4:ToyNode

N1 - e4 -> N5:ToyNode

N4 - e5 -> N5

==========

In this case, we're going to define a node type as what we're calling a *ToyNode*. As we introduce each node and its relationship with other nodes, we'll define the node to be of type *ToyNode*.

On the first line, N1 goes through e1 to N2, and both of those are introduced for the first time, so we'll define them as type ToyNode. But, on the next line, since we already introduced N2 as type ToyNode, we don't need to repeat that statement. And, so we continue in the same manner with the remaining edge relationships.

Taking this even further, we'll apply a similar kind of constraint to our edges.

==========

Even more technical pseudo-code

N1:ToyNode - ToyRelation -> N2:ToyNode

N2 - ToyRelation -> N3:ToyNode

N2 - ToyRelation -> N4:ToyNode

N1 - ToyRelation -> N5:ToyNode

N4 - ToyRelation -> N5

==========

In this case, define our network such that each edge is a particular type, which we're calling *ToyRelation*.

Next, we're going to add properties to our nodes and edges.

==========

Pseudo-code approximating CYPHER code

N1:ToyNode {name: 'Tom'} - ToyRelation {relationship: 'knows'} -> N2:ToyNode {name: 'Harry'}

N2 - ToyRelation {relationship: 'co-worker'} -> N3:ToyNode {name: 'Julian', job: 'plumber'} N2 - ToyRelation {relationship: 'wife'} -> N4:ToyNode {name: 'Michele', job: 'accountant'}

N1 - ToyRelation {relationship: 'wife'} -> N5:ToyNode {name: 'Josephine', job: 'manager'}

N4 - ToyRelation {relationship: 'friend'} -> N5

==========

Our nodes can have properties such as name or job. In this case, our first node, N1, will have the name Tom, and the appropriate syntax for this includes curly braces surrounding the key value pairs, a colon separating the key value pairs, and the values defined within single quotes. Likewise, each edge may have a specific type of relationship, including co-worker, wife, and friend.

Step 4. Develop an actual script to create our Neo4j network.

Finally, this brings us to the actual code we're going to use to create our graph network.

The actual CYPHER code to create our 'Toy' network

```
create (N1:ToyNode {name: 'Tom'}) - [:ToyRelation {relationship: 'knows'}] -> (N2:ToyNode {
(N2) - [:ToyRelation {relationship: 'co-worker'}] -> (N3:ToyNode {name: 'Julian', job: 'plu
(N2) - [:ToyRelation {relationship: 'wife'}] -> (N4:ToyNode {name: 'Michele', job: 'account
(N1) - [:ToyRelation {relationship: 'wife'}] -> (N5:ToyNode {name: 'Josephine', job: 'manag
(N4) - [:ToyRelation {relationship: 'friend'}] -> (N5)
;
```

Step 5. Run the script. Now, we are going to execute this code in Neo4j. Make sure to have your Neo4j container running and open it in your browser as we saw previously.

Copy the previous script, and paste it into Neo4j's command line:

Run the command and you will see your results in the panel.

We can see that we have 5 labels added, 5 nodes were created, 13 properties were set, 5 relationships were created, and the entire process required 28 milliseconds.

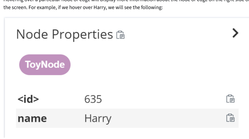
Step 6. Explore the graph network, confirm the structure and content. We still haven't seen our graph. To do that, we have to run the following command:

```
match (n:ToyNode)-[r]-(m) return n, r, m
```

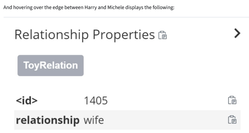
What this command does, is it tries to identify a match in which a particular node has a relationship with any other node. And, then we'll return those nodes and relationships.

Execute the command in Neo4j.

Hovering over a particular node or edge will display more information about the node or edge on the right side of the screen. For example, if we hover over Harry, we will see the following:

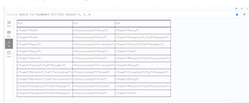


And hovering over the edge between Harry and Michele displays the following:



To keep the displaying fixed, we click on node or edge of interest.

We can also display our nodes and edges in a tabular format. Click on the *Text* button on the left:



Or we can represent our relationships more comprehensively and explore all of the properties of the nodes and edges by clicking on *Table*:

Complementary commands for experimentation

Delete all nodes and edges

```
match (n)-[r]-() delete n, r
```

Delete all nodes which have no edges

```
match (n) delete n
```

Delete only ToyNode nodes which have no edges

```
match (n:ToyNode) delete n
```

Delete all edges

```
match (n)-[r]-() delete r
```

Delete only ToyRelation edges

```
match (n)-[r:ToyRelation]-() delete r
```

Selecting an existing single ToyNode node

```
match (n:ToyNode {name:'Julian'}) return n
```