GremlinDocs

IMPORTANT - The Gremlin examples shown here refer to TinkerPop 2.x. Note that TinkerPop 3.x has been released under the Apache Software Foundation. Find the latest information for Gremlin on the Apache TinkerPop homepage.

Gremlin is a domain specific language for traversing property graphs. This language has application in the areas of graph query, analysis, and manipulation. See the Getting Started Gremlin wiki page for downloading and installing Gremlin.

Gremlin is an open source project maintained by TinkerPop. Please join the Gremlin users group at http://groups.google.com/group/gremlin-users for all TinkerPop related discussions.

Unless otherwise noted, all samples are derived from the TinkerPop "toy" graph generated with:

```
gremlin> g = TinkerGraphFactory.createTinkerGraph()
```

This produces a hardcoded representation of the graph diagrammed here.

The documentation and samples presented here attempt to stay current with the most recent, stable release of Gremlin. Please note that this is not the *official* Gremlin documentation. The official documentation resides in the Gremlin wiki. Other Gremlin documentation can be found at Sql2Gremlin, which is focused on teaching those familiar with SQL how to think in Gremlin and graphs.

GremlinDocs is a GitHub repository. Pull requests will be considered.

Acknowledgements: Gremlin artwork by Ketrina Yim and font by Maelle Keita.

Transform

Transform steps take an object and emit a transformation of it.

_

Identity turns an arbitrary object into a "pipeline".

```
gremlin> x = [1,2,3]
==>1
==>2
==>3
gremlin> x._().transform{it+1}
==>2
==>3
==>4
gremlin> x = g.E.has('weight', T.gt, 0.5f).toList()
==>e[10][4-created->5]
==>e[8][1-knows->4]
gremlin> x.inV
==>[StartPipe, InPipe]
==>[StartPipe, InPipe]
gremlin> x._().inV
==>v[5]
==>v[4]
```

both

Get both adjacent vertices of the vertex, the in and the out.

```
gremlin> v = g.v(4)
==>v[4]
gremlin> v.both
==>v[1]
==>v[5]
==>v[3]
gremlin> v.both('knows')
==>v[1]
gremlin> v.both('knows', 'created')
==>v[1]
==>v[5]
==>v[3]
gremlin> v.both(1, 'knows', 'created')
==>v[1]
```

bothE

Get both incoming and outgoing edges of the vertex.

```
gremlin> v = g.v(4)
==>v[4]
gremlin> v.bothE
==>e[8][1-knows->4]
==>e[10][4-created->5]
==>e[11][4-created->3]
gremlin> v.bothE('knows')
==>e[8][1-knows->4]
gremlin> v.bothE('knows', 'created')
==>e[8][1-knows->4]
==>e[10][4-created->5]
==>e[11][4-created->3]
gremlin> v.bothE(1,'knows','created')
==>e[8][1-knows->4]
```

bothV

Get both incoming and outgoing vertices of the edge.

```
gremlin> e = g.e(12)
==>e[12][6-created->3]
gremlin> e.outV
==>v[6]
gremlin> e.inV
==>v[3]
gremlin> e.bothV
==>v[6]
==>v[3]
```

cap

Gets the side-effect of the pipe prior. In other words, it emits the value of the previous step and not the values that flow through it.

```
gremlin> g.V('lang', 'java').in('created').name.groupCount
==>marko
==>josh
==>peter
==>josh
gremlin> g.V('lang', 'java').in('created').name.groupCount.cap
==>{marko=1, peter=1, josh=2}
```

E

The edge iterator for the graph. Utilize this to iterate through all the edges in the graph. Use with care on large graphs.

```
gremlin> g.E
==>e[10][4-created->5]
==>e[7][1-knows->2]
==>e[9][1-created->3]
==>e[8][1-knows->4]
==>e[11][4-created->3]
==>e[12][6-created->3]
gremlin> g.E.weight
==>1.0
==>0.5
==>0.4
==>1.0
==>0.4
==>0.2
```

gather

Collect all objects up to that step and process the gathered list with the provided closure.

```
gremlin> g.v(1).out
==>v[2]
==>v[4]
==>v[3]
gremlin> g.v(1).out.gather
==>[v[2], v[4], v[3]]
gremlin> g.v(1).out.gather{it.size()}
==>3
```

See Also

id

Gets the unique identifier of the element.

```
gremlin> v = g.V("name", "marko").next()
==>v[1]
gremlin> v.id
==>1
gremlin> g.v(1).id
==>1
```

in

Gets the adjacent vertices to the vertex.

```
gremlin> v = g.v(4)
==>v[4]
gremlin> v.inE.outV
==>v[1]
gremlin> v.in
==>v[1]
gremlin> v = g.v(3)
==>v[3]
gremlin> v.in("created")
==>v[1]
==>v[4]
==>v[6]
gremlin> v.in(2,'created')
==>v[1]
==>v[4]
gremlin> v.inE("created").outV
==>v[1]
==>v[4]
==>v[6]
gremlin> v.inE(2,'created').outV[0]
==>v[1]
```

inE

Gets the incoming edges of the vertex.

```
gremlin> v = g.v(4)
==>v[4]
gremlin> v.inE.outV
==>v[1]
gremlin> v.in
==>v[1]
gremlin> v = g.v(3)
==>v[3]
gremlin> v.in("created")
==>v[1]
==>v[4]
==>v[6]
gremlin> v.in(2,'created')
==>v[1]
```

```
==>v[4]
gremlin> v.inE("created").outV
==>v[1]
==>v[4]
==>v[6]
gremlin> v.inE(2,'created').outV[0]
==>v[1]
```

inV

Get both incoming head vertex of the edge.

```
gremlin> e = g.e(12)
==>e[12][6-created->3]
gremlin> e.outV
==>v[6]
gremlin> e.inV
==>v[3]
gremlin> e.bothV
==>v[6]
==>v[3]
```

key

Get the property value of an element. The property value can be obtained by simply appending the name to the end of the element or by referencing it as a Groovy map element with square brackets. For best performance, drop down to the Blueprints API and use `getProperty(key)`.

```
gremlin> v = g.v(3)
==>v[3]
gremlin> v.name
==>lop
gremlin> v['name']
==>lop
gremlin> x = 'name'
==>name
gremlin> v[x]
==>lop
gremlin> v.getProperty('name')
==>lop
```

label

Gets the label of an edge.

```
gremlin> g.v(6).outE.label
==>created
gremlin> g.v(1).outE.filter{it.label=='created'}
==>e[9][1-created->3]

// a more efficient approach to use of label
gremlin> g.v(1).outE.has('label','created')
==>e[9][1-created->3]
```

linkBoth[In/Out]

An element-centric mutation that takes every incoming vertex and creates an edge to the provided vertex. It can be used with both a Vertex object or a named step.

```
gremlin> marko = g.v(1)
==>v[1]
gremlin> g.V.except([marko]).linkBoth('connected',marko)
==>v[3]
==>v[2]
==>v[6]
==>v[5]
==>v[4]
gremlin> marko.outE('connected')
==>e[2][1-connected->2]
==>e[0][1-connected->3]
==>e[6][1-connected->5]
==>e[4][1-connected->6]
==>e[14][1-connected->4]
gremlin> g.V.except([marko]).outE('connected')
==>e[1][3-connected->1]
==>e[3][2-connected->1]
==>e[5][6-connected->1]
==>e[13][5-connected->1]
==>e[15][4-connected->1]
gremlin>
g.v(1).as('x').out('created').in('created').except([g.v(1)]).linkBoth('coc
reator','x')
==>v[4]
==>v[6]
gremlin> g.E.has('label','cocreator')
==>e[3][6-cocreator->1]
==>e[2][1-cocreator->6]
==>e[1][4-cocreator->1]
==>e[0][1-cocreator->4]
```

map

Gets the property map of the graph element.

```
gremlin> g.v(1).map
==>{name=marko, age=29}
gremlin> g.v(1).map()
==>name=marko
==>age=29
gremlin> g.V.map('id','age')
==>{id=3, age=null}
==>{id=2, age=27}
==>{id=1, age=29}
==>{id=6, age=35}
==>{id=5, age=null}
==>{id=4, age=32}
gremlin> g.v(1)._().map('id','age')
==>{id=1, age=29}
```

memoize

Remembers a particular mapping from input to output. Long or expensive expressions with no side effects can use this step to remember a mapping, which helps reduce load when previously processed objects are passed into it.

For situations where memoization may consume large amounts of RAM, consider using an embedded key-value store like JDBM or some other persistent Map implementation.

```
gremlin> g.V.out.out.memoize(1).name
==>ripple
==>lop
gremlin> g.V.out.as('here').out.memoize('here').name
==>ripple
==>lop
gremlin> m = [:]
gremlin> g.V.out.out.memoize(1,m).name
==>ripple
==>lop
```

order

Order the items in the stream according to the closure if provided. If no closure is provided, then a default sort order is used.

```
gremlin> g.V.name.order
==>josh
==>lop
==>marko
==>peter
==>ripple
==>vadas
gremlin>  g.V.name.order{it.b <=> it.a}
==>vadas
==>ripple
==>peter
==>marko
==>lop
==>josh
gremlin> g.V.order{it.b.name <=> it.a.name}.out('knows')
==>v[2]
==>v[4]
```

orderMap

For every incoming map, sort with supplied closure or `T.decr` or `T.incr` and emit keys.

```
gremlin> g.V.both.groupCount.cap.next()
==>v[3]=3
==>v[2]=1
==>v[1]=3
==>v[6]=1
==>v[5]=1
==>v[4]=3
gremlin> g.V.both.groupCount.cap.orderMap(T.decr)
==>v[3]
==>v[1]
==>v[4]
==>v[2]
==>v[6]
==>v[5]
gremlin> g.V.both.groupCount.cap.orderMap(T.decr)[0..1]
==>v[3]
==>v[1]
gremlin> g.V.both.groupCount.cap.orderMap(T.decr)[0..1].name
==>lop
==>marko
```

out

Gets the out adjacent vertices to the vertex.

```
gremlin> v = g.v(1)
==>v[1]
gremlin> v.outE.inV
==>v[2]
==>v[4]
==>v[3]
gremlin> v.out
==>v[2]
==>v[4]
==>v[3]
gremlin> v.outE('knows').inV
==>v[2]
==>v[4]
gremlin> v.out('knows')
==>v[2]
==>v[4]
gremlin> v.out(1,'knows')
==>v[2]
```

outE

Gets the outgoing edges to the vertex.

```
gremlin> v = g.v(1)
==>v[1]
gremlin> v.outE.inV
==>v[2]
==>v[4]
==>v[3]
gremlin> v.outE
==>e[7][1-knows->2]
==>e[8][1-knows->4]
==>e[9][1-created->3]
gremlin> v.outE('knows').inV
==>v[2]
==>v[4]
gremlin> v.outE('knows')
==>e[7][1-knows->2]
==>e[8][1-knows->4]
gremlin> v.outE(1,'knows')
```

```
==>e[7][1-knows->2]
```

outV

Get both outgoing tail vertex of the edge.

```
gremlin> e = g.e(12)
==>e[12][6-created->3]
gremlin> e.outV
==>v[6]
gremlin> e.inV
==>v[3]
gremlin> e.bothV
==>v[6]
==>v[3]
```

path

Gets the path through the pipeline up to this point, where closures are post-processing for each object in the path. If the path step is provided closures then, in a round robin fashion, the closures are evaluated over each object of the path and that post-processed path is returned.

```
gremlin> g.v(1).out.path
==>[v[1], v[2]]
==>[v[1], v[4]]
==>[v[1], v[3]]
gremlin> g.v(1).out.path{it.id}
==>[1, 2]
==>[1, 4]
==>[1, 3]
gremlin> g.v(1).out.path{it.id}{it.name}
==>[1, vadas]
==>[1, josh]
==>[1, lop]
gremlin> g.v(1).outE.inV.name.path
==>[v[1], e[7][1-knows->2], v[2], vadas]
==>[v[1], e[8][1-knows->4], v[4], josh]
==>[v[1], e[9][1-created->3], v[3], lop]
```

See Also

scatter

Unroll all objects in the iterable at that step. Gather/Scatter is good for breadth-first traversals where the gather closure filters out unwanted elements at the current radius.

```
gremlin> g.v(1).out
==>v[2]
==>v[4]
==>v[3]
gremlin> g.v(1).out.gather{it[1..2]}
==>[v[4], v[3]]
gremlin> g.v(1).out.gather{it[1..2]}.scatter
==>v[4]
==>v[3]
```

See Also

select

Select the named steps to emit after select with post-processing closures.

```
gremlin> g.v(1).as('x').out('knows').as('y').select
==>[x:v[1], y:v[2]]
==>[x:v[1], y:v[4]]
gremlin> g.v(1).as('x').out('knows').as('y').select(["y"])
==>[y:v[2]]
==>[y:v[4]]
gremlin> g.v(1).as('x').out('knows').as('y').select(["y"]){it.name}
==>[y:vadas]
==>[y:josh]
gremlin>  g.v(1).as('x').out('knows').as('y').select{it.id}{it.name}
==>[x:1, y:vadas]
==>[x:1, y:josh]
```

shuffle

Collect all objects up to that step into a list and randomize their order.

```
gremlin> g.v(1).out.shuffle
==>v[2]
==>v[3]
==>v[4]
gremlin> g.v(1).out.shuffle
==>v[3]
==>v[2]
==>v[4]
```

See Also

transform

Transform emits the result of a closure.

```
gremlin> g.E.has('weight', T.gt, 0.5f).outV.age
==>32
==>29
gremlin> g.E.has('weight', T.gt, 0.5f).outV.age.transform{it+2}
==>34
==>31
gremlin> g.E.has('weight', T.gt, 0.5f).outV.transform{[it.id,it.age]}
==>[4, 32]
==>[1, 29]
gremlin> g.E.has('weight', T.gt,
0.5f).outV.transform{[id:it.id,age:it.age]}
==>{id=4, age=32}
==>{id=1, age=29}
```

V

The vertex iterator for the graph. Utilize this to iterate through all the vertices in the graph. Use with care on large graphs unless used in combination with a key index lookup.

```
gremlin> g.V
==>v[3]
==>v[2]
==>v[1]
==>v[6]
==>v[5]
==>v[4]
gremlin> g.V("name", "marko")
==>v[1]
gremlin> g.V("name", "marko").name
==>marko
```

Filter

Filter steps decide whether to allow an object to pass to the next step or not.

[i]

A index filter that emits the particular indexed object.

```
gremlin> g.V[0].name
==>lop
```

[i..j]

A range filter that emits the objects within a range.

```
gremlin> g.V[0..2].name
==>lop
==>vadas
==>marko
gremlin> g.V[0..<2].name
==>lop
==>vadas
```

and

Takes a collection of pipes and emits incoming objects that are true for all of the pipes.

```
gremlin> g.v(1).outE.and(_().has('weight', T.gt, 0.4f), _().has('weight', T.lt, 0.8f))
==>e[7][1-knows->2]
gremlin> g.V.and(_().both("knows"), _().both("created"))
==>v[1]
==>v[4]
```

back

Go back to the results of a named step.

```
gremlin> g.V.as('x').outE('knows').inV.has('age', T.gt, 30).back('x').age
==>29
```

dedup

Emit only incoming objects that have not been seen before with an optional closure being the object to check on.

```
gremlin> g.v(1).out.in
==>v[1]
==>v[1]
==>v[1]
==>v[4]
==>v[6]
```

```
gremlin> g.v(1).out.in.dedup()
==>v[1]
==>v[4]
==>v[6]
```

except

Emit everything to pass except what is in the supplied collection or in the results of a named step.

```
gremlin> x = [g.v(1), g.v(2), g.v(3)]
==>v[1]
==>v[2]
==>v[3]
gremlin> g.V.except(x)
==>v[6]
==>v[5]
==>v[4]
gremlin> x = []
gremlin> g.v(1).out.aggregate(x).out.except(x)
==>v[5]
gremlin>
g.V.has('age',T.lt,30).as('x').out('created').in('created').except('x')
==>v[4]
==>v[6]
```

See Also

filter

Decide whether to allow an object to pass. Return true from the closure to allow an object to pass.

```
gremlin> g.V.filter{it.age > 29}.name
==>peter
==>josh
```

has

Allows an element if it has a particular property. Utilizes several options for comparisons through `T`:

- T.gt - greater than
- T.gte - greater than or equal to
- T.eq - equal to
- T.neq - not equal to
- T.lte - less than or equal to

- T.lt - less than
- T.in - contained in a list
- T.notin - not contained in a list

It is worth noting that the syntax of `has` is similar to `g.V("name", "marko")`, which has the difference of being a key index lookup and as such will perform faster. In contrast, this line, `g.V.has("name", "marko")`, will iterate over all vertices checking the `name` property of each vertex for a match and will be significantly slower than the key index approach. All that said, the behavior of `has` is dependent upon the underlying implementation and the above description is representative of *most* Blueprints implementations. For instance, Titan will actually try to use indices where it sees the opportunity to do so. It is therefore important to understand the functionality of the underlying database when writing traversals.

```
gremlin> g.V.has("name", "marko").name
==>marko
gremlin> g.v(1).outE.has("weight", T.gte, 0.5f).weight
==>0.5
==>1.0
gremlin> g.V.has('age').name
==>vadas
==>marko
==>peter
==>josh
gremlin> g.V.has('age',T.in,[29,32])
==>v[1]
==>v[4]
gremlin> g.V.has('age').has('age',T.notin, [27,35]).name
==>marko
==>josh
```

See Also

hasNot

Allows an element if it does not have a particular property. Utilizes several options for comparisons on through `T`:

- T.gt - greater than
- T.gte - greater than or equal to
- T.eq - equal to
- T.neq - not equal to
- T.lte - less than or equal to
- T.lt - less than
- T.in - contained in a list

- T.notin - not contained in a list

```
gremlin> g.v(1).outE.hasNot("weight", T.eq, 0.5f).weight
==>1.0
==>0.4
gremlin> g.V.hasNot('age').name
==>lop
==>ripple
```

interval

Allow elements to pass that have their property in the provided start and end interval.

```
gremlin> g.E.interval("weight", 0.3f, 0.9f).weight
==>0.5
==>0.4
==>0.4
```

See Also

or

Takes a collection of pipes and emits incoming objects that are true for any of the pipes.

```
gremlin> g.v(1).outE.or(_().has('id', T.eq, "9"), _().has('weight', T.lt,
0.6f))
==>e[7][1-knows->2]
==>e[9][1-created->3]
```

random

Emits the incoming object if biased coin toss is heads.

```
gremlin> g.V.random(0.5)
==>v[3]
==>v[1]
==>v[6]
gremlin> g.V.random(0.5)
==>v[2]
==>v[5]
==>v[4]
```

retain

Allow everything to pass except what is not in the supplied collection or in the results of a named step.

```
gremlin> x = [g.v(1), g.v(2), g.v(3)]
==>v[1]
==>v[2]
==>v[3]
gremlin> g.V.retain(x)
==>v[3]
==>v[2]
==>v[1]
gremlin> x = []
gremlin> g.v(1).out.aggregate(x).out.retain(x)
==>v[3]
gremlin> g.V.as('x').both.both.both.retain('x')
==>v[3]
==>v[3]
==>v[1]
==>v[1]
==>v[4]
==>v[4]
```

See Also

simplePath

Emit the object only if the current path has no repeated elements.

```
gremlin> g.v(1).out.in
==>v[1]
==>v[1]
==>v[1]
==>v[4]
==>v[6]
gremlin> g.v(1).out.in.simplePath
==>v[4]
==>v[6]
```

Side Effect

Side Effect steps pass the object, but yield some kind of side effect while doing so.

aggregate

Emits input, but adds input in collection, where provided closure processes input prior to insertion (greedy). In being "greedy", 'aggregate' will exhaust all the items that come to it from previous steps before emitting the next element.

```
gremlin> x = []
gremlin> g.v(1).out.aggregate(x).next()
==>v[2]
gremlin> x
==>v[2]
==>v[4]
==>v[3]
```

See Also

as

Emits input, but names the previous step.

```
gremlin> g.V.as('x').outE('knows').inV.has('age', T.gt, 30).back('x').age
==>29
```

groupBy

Emits input, but groups input after processing it by provided key-closure and value-closure. It is also possible to supply an optional reduce-closure.

```
gremlin> g.V.groupBy{it}{it.out}.cap
==>{v[3]=[], v[2]=[], v[1]=[v[2], v[4], v[3]], v[6]=[v[3]], v[5]=[], v[4]=
[v[5], v[3]]}
gremlin> g.V.groupBy{it}{it.out}{it.size()}.cap
==>{v[3]=0, v[2]=0, v[1]=3, v[6]=1, v[5]=0, v[4]=2}
gremlin> m = [:]
gremlin> g.V.groupBy(m){it}{it.out}.iterate();null;
==>null
gremlin> m
==>v[3]=[]
==>v[2]=[]
==>v[1]=[v[2], v[4], v[3]]
==>v[6]=[v[3]]
==>v[5]=[]
==>v[4]=[v[5], v[3]]
gremlin> g.V.out.groupBy{it.name}{it.in}{it.unique().findAll{i -> i.age >
30}.name}.cap
==>{lop=[josh, peter], ripple=[josh], josh=[], vadas=[]}
```

See Also

groupCount

Emits input, but updates a map for each input, where closures provides generic map update.

```
gremlin> g.V.out.groupCount(m)
==>v[2]
==>v[4]
==>v[3]
==>v[3]
==>v[5]
==>v[3]
gremlin> m
==>v[2]=1
==>v[4]=1
==>v[3]=3
==>v[5]=1
gremlin> g.v(1).out.groupCount(m){it}{it.b+1.0}.out.groupCount(m){it}
{it.b+0.5}
==>v[5]
==>v[3]
gremlin> m
==>v[2]=1.0
==>v[4]=1.0
==>v[5]=0.5
==>v[3]=1.5
```

See Also

optional

Behaves similar to `back` except that it does not filter. It will go down a particular path and back up to where it left off. As such, its useful for yielding a side-effect down a particular branch.

```
gremlin> g.V.as('x').outE('knows').inV.has('age', T.gt, 30).back('x')
==>v[1]
gremlin> g.V.as('x').outE('knows').inV.has('age', T.gt, 30).optional('x')
==>v[3]
==>v[2]
==>v[1]
==>v[6]
==>v[5]
==>v[4]
```

See Also

sideEffect

Emits input, but calls a side effect closure on each input.

```
gremlin> youngest = Integer.MAX_VALUE
==>2147483647
gremlin> g.V.has('age').sideEffect{youngest=youngest>it.age?
it.age:youngest}
==>v[2]
==>v[1]
==>v[6]
==>v[4]
gremlin> youngest
==>27
```

store

Emits input, but adds input to collection, where provided closure processes input prior to insertion (lazy). In being "lazy", 'store' will keep element as they are being requested.

```
gremlin> x = []
gremlin> g.v(1).out.store(x).next()
==>v[2]
gremlin> x
==>v[2]
```

See Also

Emit input, but stores the tree formed by the traversal as a map. Accepts an optional set of closures to be applied in round-robin fashion over each level of the tree.

```
gremlin> g.v(1).out.out.tree.cap
==>{v[1]={v[4]={v[3]={}, v[5]={}}}}
gremlin> g.v(1).out.out.tree{it.name}.cap
==>{marko={josh={lop={}, ripple={}}}}
gremlin> g.v(1).out.out.tree{it.name}{"child1:" + it.name}{"child2:" +
it.name}.cap
==>{marko={child1:josh={child2:lop={}, child2:ripple={}}}}
gremlin> t = new Tree()
gremlin> g.v(1).out.out.tree(t){it.name}{"child1:" + it.name}{"child2:" +
it.name}
==>v[5]
==>v[3]
```

```
gremlin> t.get('marko')
==>child1:josh={child2:lop={}, child2:ripple={}}
```

Branch

Branch steps decide which step to take.

copySplit

Copies incoming object to internal pipes.

```
gremlin> g.v(1).out('knows').copySplit(_().out('created').name,
_().age).fairMerge
==>ripple
==>27
==>lop
==>32
gremlin> g.v(1).out('knows').copySplit(_().out('created').name,
_().age).exhaustMerge
==>ripple
==>lop
==>27
==>32
```

See Also

exhaustMerge

Used in combination with a `copySplit`, merging the parallel traversals by exhaustively getting the objects of the first, then the second, etc.

```
gremlin> g.v(1).out('knows').copySplit(_().out('created').name,
_().age).exhaustMerge
==>ripple
==>lop
==>27
==>32
```

See Also

fairMerge

Used in combination with a `copySplit`, merging the parallel traversals in a round-robin fashion.

```
gremlin> g.v(1).out('knows').copySplit(_().out('created').name,
_().age).fairMerge
==>ripple
==>27
==>lop
==>32
```

See Also

ifThenElse

Allows for if-then-else conditional logic.

```
gremlin> g.v(1).out.ifThenElse{it.name=='josh'}{it.age}{it.name}
==>vadas
==>32
==>lop
```

loop

Loop over a particular set of steps in the pipeline. The first argument is either the number of steps back in the pipeline to go or a named step. The second argument is a while closure evaluating the current object. The `it` component of the loop step closure has three properties that are accessible. These properties can be used to reason about when to break out of the loop.

- `it.object`: the current object of the traverser.
- `it.path`: the current path of the traverser.
- `it.loops`: the number of times the traverser has looped through the loop section.

The final argument is known as the "emit" closure. This boolean-based closure will determine wether the current object in the loop structure is emitted or not. As such, it is possible to emit intermediate objects, not simply those at the end of the loop.

```
gremlin> g.v(1).out.out
==>v[5]
==>v[3]
gremlin> g.v(1).out.loop(1){it.loops<3}
==>v[5]
==>v[3]
gremlin> g.v(1).out.loop(1){it.loops<3}{it.object.name=='josh'}
==>v[4]
```

Methods

Methods represent functions that make it faster and easier to work with Blueprints and Pipes APIs. It is important to keep in mind that the full Java API and Groovy API are accessible from Gremlin.

Element.keys

Get the property keys of an element.

```
gremlin> g.v(1).keys()
==>name
==>age
```

Element.remove

Remove an element from the graph.

```
gremlin> g.E.weight
==>1.0
==>0.5
==>0.4
==>1.0
==>0.4
==>0.2
gremlin> g.E.has("weight",T.lt,0.5f).remove()
==>null
gremlin> g.E.weight
==>1.0
==>0.5
==>1.0
```

Element.values

Gets the property values of an element.

```
gremlin> g.v(1).values()
==>marko
==>29
```

Graph.addEdge

Adds an edge to the graph. Note that most graph implementations ignore the identifier supplied to `addEdge`.

```
gremlin> g = new TinkerGraph()
==>tinkergraph[vertices:0 edges:0]
```

```
gremlin> v1 = g.addVertex(100)
==>v[100]
gremlin> v2 = g.addVertex(200)
==>v[200]
gremlin> g.addEdge(v1,v2,'friend')
==>e[0][100-friend->200]
gremlin> g.addEdge(1000,v1,v2,'buddy')
==>e[1000][100-buddy->200]
gremlin> g.addEdge(null,v1,v2,'pal',[weight:0.75f])
==>e[1][100-pal->200]
```

Graph.addVertex

Adds a vertex to the graph. Note that most graph implementations ignore the identifier supplied to `addVertex`.

```
gremlin> g = new TinkerGraph()
==>tinkergraph[vertices:0 edges:0]
gremlin> g.addVertex()
==>v[0]
gremlin> g.addVertex(100)
==>v[100]
gremlin> g.addVertex(null,[name:"stephen"])
==>v[1]
```

Graph.e

Get an edge or set of edges by providing one or more edge identifiers. The identifiers must be the identifiers assigned by the underlying graph implementation.

```
gremlin> g.e(10)
==>e[10][4-created->5]
gremlin> g.e(10,11,12)
==>e[10][4-created->5]
==>e[11][4-created->3]
==>e[12][6-created->3]
gremlin> ids = [10,11,12]
==>10
==>11
==>12
gremlin> g.e(ids.toArray())
==>e[10][4-created->5]
==>e[11][4-created->3]
==>e[12][6-created->3]
```

See Also

Graph.idx(String)

Get an manual index by its name.

```
gremlin> g.createIndex("my-index", Vertex.class)
==>index[my-index:Vertex]
gremlin> myIdx = g.idx("my-index").put("name", "marko", g.v(1))
==>null
gremlin> myIdx.getIndexName()
==>my-index
```

See Also

Graph.load

Load a file from one of several standard formats such as GraphML, GML, or GraphSON.

```
gremlin> g = new TinkerGraph()
==>tinkergraph[vertices:0 edges:0]
gremlin> g.loadGraphML('data/graph-example-1.xml')
==>null
gremlin> g.V
==>v[3]
==>v[2]
==>v[1]
==>v[6]
==>v[5]
==>v[4]
```

See Also

Graph.removeEdge

Remove an edge.

```
gremlin> g = new TinkerGraph()
==>tinkergraph[vertices:0 edges:0]
gremlin> v1 = g.addVertex()
==>v[100]
gremlin> v2 = g.addVertex()
==>v[200]
gremlin> g.addEdge(v1,v2,'friend')
==>e[0][100-friend->200]
```

```
gremlin> g.removeEdge(g.e(0))
==>null
```

Graph.removeVertex

Remove a vertex.

```
gremlin> g.addVertex()
==>v[128]
gremlin> g.removeVertex(g.v(128))
==>null
```

Graph.save

Save a graph to file given one of several standard formats such as GraphML, GML, or GraphSON.

```
gremlin> g.saveGraphML('data/graph.xml')
==>null
```

See Also

Graph.v

Get a vertex or set of vertices by providing one or more vertex identifiers. The identifiers must be the identifiers assigned by the underlying graph implementation.

```
gremlin> g.v(1)
==>v[1]
gremlin> g.v(1,2,3)
==>v[1]
==>v[2]
==>v[3]
gremlin> ids = [1,2,3]
==>1
==>2
==>3
gremlin> g.v(ids.toArray())
==>v[1]
==>v[2]
==>v[3]
```

See Also

Index[Map.Entry]

Look up a value in an index.

```
gremlin> g.createIndex("my-index", Vertex.class)
==>index[my-index:Vertex]
gremlin> g.idx("my-index").put("name", "marko", g.v(1))
==>null
gremlin> g.idx("my-index")[[name:"marko"]]
==>v[1]
```

See Also

Pipe.enablePath

If the path information is required internal to a closure, Gremlin doesn't know that as it can not interpret what is in a closure. As such, be sure to use `GremlinPipeline.enablePath()` if path information will be required by the expression.

```
gremlin> g.v(1).out.loop(1){it.loops < 3}{it.path.contains(g.v(4))}
Cannot invoke method contains() on null object
Display stack trace? [yN]
gremlin> g.v(1).out.loop(1){it.loops < 3}
{it.path.contains(g.v(4))}.enablePath()
==>v[5]
==>v[3]
```

See Also

Pipe.fill

Takes all the results in the pipeline and puts them into the provided collection.

```
gremlin> m = []
gremlin> g.v(1).out.fill(m)
==>v[2]
==>v[4]
==>v[3]
gremlin> m
==>v[2]
==>v[4]
==>v[3]
```

See Also

Pipe.iterate

Calls Pipe.next for all objects in the pipe. This is an important notion to follow when considering the behavior of the Gremlin Console. The Gremlin Console iterates through the pipeline automatically and outputs the results. Outside of the Gremlin Console or if more than one statement is present on a single line of the Gremlin Console, iterating the pipe must be done manually. Read more about this topic in the Gremlin Wiki Troubleshooting Page.

There are some important things to note in the example below. Had the the first line of Gremlin been executed separately, as opposed to being placed on the same line separated by a semi-colon, the names of all the vertices would have changed because the Gremlin Console would have automatically iterated the pipe and processed the side-effects.

```
gremlin> g.V.sideEffect{it.name="same-again"};g.V.name
==>lop
==>vadas
==>marko
==>peter
==>ripple
==>josh
gremlin> g.V.sideEffect{it.name="same"}.iterate();g.V.name
==>same
==>same
==>same
==>same
==>same
==>same
```

See Also

Pipe.next

Gets the next object in the pipe or the next *n* objects. This is an important notion to follow when considering the behavior of the Gremlin Console. The Gremlin Console iterates through the pipeline automatically and outputs the results. Outside of the Gremlin Console or if more than one statement is present on a single line of the Gremlin Console, iterating the pipe must be done manually. Read more about this topic in the Gremlin Wiki Troubleshooting Page.

There are some important things to note in the example below. Had the the first line of Gremlin been executed separately, as opposed to being placed on the same line separated by a semi-colon, the name of the vertex would have changed because the Gremlin Console would have automatically iterated the pipe and processed the side-effect.

```
gremlin> g.v(1).sideEffect{it.name="same"};g.v(1).name
==>marko
gremlin> g.v(1).sideEffect{it.name="same"}.next();g.v(1).name
```

```
==>same
gremlin> g.V.sideEffect{it.name="same-again"}.next(3);g.V.name
==>same-again
==>same-again
==>same-again
==>peter
==>ripple
==>josh
```

See Also

Recipes

Recipes are common patterns that are seen in using Gremlin.

Duplicate Edges

Strictly speaking, you cannot have duplicated egdes with the same id. This example finds edges with same `outV/inV/label` properties.

```
gremlin> g = TinkerGraphFactory.createTinkerGraph()
==>tinkergraph[vertices:6 edges:6]
gremlin> g.v(1).outE('created')
==>e[9][1-created->3]
gremlin> g.addEdge(null, g.v(1), g.v(3), "created", g.e(9).map()) // see
note
==>e[0][1-created->3]
gremlin> g.v(1).outE('created')
==>e[0][1-created->3]
==>e[9][1-created->3]
gremlin> ElementHelper.haveEqualProperties(g.e(9), g.e(0))
==>true
gremlin> e = g.e(9)
==>e[9][1-created->3]
gremlin>
e.outV.outE(e.label).filter{ElementHelper.haveEqualProperties(e,it)}.as('e
').inV.filter{it==e.inV.next()}.back('e').except([e])
==>e[0][1-created->3]
```

Finding Edges Between Vertices

It is often useful to determine if there is an edge between one vertex and another.

```
gremlin> g.v(1).both.retain([g.v(3)]).hasNext()
```

```
==>true
gremlin> g.v(1).bothE.as('x').bothV.retain([g.v(3)]).back('x')
==>e[9][1-created->3]
```

Happy Birthday

Some Gremlin for that special occassion when you don't know what other gift to give.

```
g = new TinkerGraph()
v01 = g.addVertex(["UC":"B","i":2]); v02 = g.addVertex(["UC":"H","i":1])
v03 = g.addVertex(["LC":"a"]); v04 = g.addVertex(["LC":"a"]);
v05 = g.addVertex(["LC":"d"]); v06 = g.addVertex(["LC":"h"]);
v07 = g.addVertex(["LC":"i"]); v08 = g.addVertex(["LC":"p"]);
v09 = g.addVertex(["LC":"p"]); v10 = g.addVertex(["LC":"r"]);
v11 = g.addVertex(["LC":"t"]); v12 = g.addVertex(["LC":"y"]);
v13 = g.addVertex(["LC":"y"]); v14 = g.addVertex(["LC":"!"]);
v02.addEdge("followedBy", v03); v03.addEdge("followedBy", v08);
v08.addEdge("followedBy", v09); v09.addEdge("followedBy", v12);
v01.addEdge("followedBy", v07); v07.addEdge("followedBy", v10);
v10.addEdge("followedBy", v11); v11.addEdge("followedBy", v06);
v06.addEdge("followedBy", v05); v05.addEdge("followedBy", v04);
v04.addEdge("followedBy", v13); v13.addEdge("followedBy", v14);

g.V().has("UC").order({ it.a.i <=> it.b.i }).transform({
  it.as("x").out("followedBy").loop("x", {true},
{true}).path().toList().reverse()[0]._().transform({ it.UC ?: it.LC
}).join()
}).join(" ")
```

Hiding Console Output

The Gremlin Console automatically iterates the pipe and outputs the results to the console. In some cases, this can lead to lots of screen output that isn't terribly useful. To suppress the output, consider the following:

```
gremlin> g.V.sideEffect{it.name='changed'}
==>v[3]
==>v[2]
==>v[1]
==>v[6]
==>v[5]
==>v[4]
gremlin> g.V.sideEffect{it.name='changed-again'}.iterate()
==>null
```

```
gremlin> g.V.name
==>changed-again
==>changed-again
==>changed-again
==>changed-again
==>changed-again
==>changed-again
gremlin> t = g.v(1).out.tree.cap.next()
==>v[1]={v[3]={}, v[2]={}, v[4]={}}
gremlin> t
==>v[1]={v[3]={}, v[2]={}, v[4]={}}
gremlin> s = g.v(1).out.tree.cap.next();null
==>null
gremlin> s
==>v[1]={v[3]={}, v[2]={}, v[4]={}}
```

See Also

Paging Results

It is sometimes desireable to not return an entire results set. Results can be paged or limited as follows:

```
gremlin> g.V.has("age", T.gte, 25)
==>v[2]
==>v[1]
==>v[6]
==>v[4]
gremlin> g.V.has("age", T.gte, 29)
==>v[1]
==>v[6]
==>v[4]
gremlin> g.V.has("age", T.gte, 29)[0..1]
==>v[1]
==>v[6]
gremlin> g.V.has("age", T.gte, 29)[0..<1]
==>v[1]
gremlin> g.V.has("age", T.gte, 29)[1..2]
==>v[6]
==>v[4]
```

Reading From a File

Reading data from an edge file formatted as CSV is easy to do with Gremlin.

```
gremlin> g = new TinkerGraph()
==>tinkergraph[vertices:0 edges:0]
gremlin> vs=[] as Set;new
File("edges.txt").eachLine{l->p=l.split(",");vs<<p[0];vs<<p[1];}
==>1
==>2
==>3
==>4
gremlin> vs.each{v->g.addVertex(v)}
==>1
==>2
==>3
==>4
gremlin> new File("edges.txt").eachLine{l-
>p=l.split(",");g.addEdge(g.getVertex(p[0]),g.getVertex(p[1]),'friend')}
gremlin> g.E
==>e[3][1-friend->4]
==>e[2][3-friend->4]
==>e[1][2-friend->3]
==>e[0][1-friend->2]
```

Sampling

It is sometimes useful to grab a random sample of the items in a collection. That can be done to some degree with the random step, but getting an explicit number of items is not supported using that step.

```
gremlin> g.v(1).out.shuffle
==>v[2]
==>v[3]
==>v[4]
gremlin> g.v(1).out.shuffle
==>v[3]
==>v[2]
==>v[4]
gremlin> g.v(1).out.random(0.5)
==>v[2]
gremlin> g.v(1).out.random(0.5)
==>v[4]
==>v[3]
```

Shortest Path

Finding the shortest path between two vertices can be accomplished with a loop. The following example shows the shortest path between vertex `1` and vertex `5` and if such path cannot be found in five steps, break out of the computation.

```
gremlin> g.v(1).out.loop(1){it.object.id != "5" && it.loops < 6}.path
==>[v[1], v[4], v[5]]
```

In the event there are multiple paths to `5`, all paths will be output and the shortest path will need to be selected. The following example adds some edges to the toy graph to demonstrate a path length distribution:

```
gremlin> g.addEdge(g.v(3), g.v(5), 'created')
==>e[0][3-created->5]
gremlin> g.addEdge(g.v(1), g.v(5), 'created')
==>e[1][1-created->5]
gremlin> g.v(1).out.loop(1){it.object.id!="5" && it.loops <
6}.path.filter{it.last().id=="5"}.transform{it.name}
==>[marko, ripple]
==>[marko, josh, ripple]
==>[marko, lop, ripple]
==>[marko, josh, lop, ripple]
gremlin> g.v(1).out.loop(1){it.object.id!="5" && it.loops <
6}.path.filter{it.last().id=="5"}.transform{it.name}.groupBy{it.size()}
{it}.cap.next()
==>2=[[marko, ripple]]
==>3=[[marko, josh, ripple], [marko, lop, ripple]]
==>4=[[marko, josh, lop, ripple]]
```

Starting with a new "toy" TinkerGraph, calculating the "cost" of the path (in this case utilizing the `weight` stored on the edges), can be accomplished with:

```
gremlin> g = TinkerGraphFactory.createTinkerGraph()
==>tinkergraph[vertices:6 edges:6]
gremlin> g.v(1).outE.inV.loop(2){it.object.id!="3" && it.loops <
6}.path.filter{it.last().id=="3"}.transform{[it.findAll{it instanceof
Edge}.sum{it.weight}, it]}
==>[0.4, [v[1], e[9][1-created->3], v[3]]]
==>[1.4000000059604645, [v[1], e[8][1-knows->4], v[4], e[11][4-created-
>3], v[3]]]
```

Evaluating a shortest path between two vertices in both directions (in and out edges) can be expensive. Consider this example:

```
gremlin> g = new TinkerGraph()
==>tinkergraph[vertices:0 edges:0]
```

```
gremlin>
gremlin> root = g.addVertex()
==>v[0]
gremlin>
gremlin> (1..10).each { outer ->
gremlin>   parent = root
gremlin>   (1..10).each { inner ->
gremlin>     child = g.addVertex()
gremlin>     g.addEdge(parent, child, 'to')
gremlin>     parent = child
gremlin>   }
gremlin> }; null
==>null
```

Given the sample graph generated above, the following uses the same pattern shown above for finding the shortest path, but evaulates both incoming and outgoing edges (prior examples only evaluated outgoing edges).

```
gremlin> target = '99'; c = 0; root.both().sideEffect{c++}.loop(2)
{it.object.id != target && it.loops <=
10}.has('id',target).path().iterate(); c
==>557670
```

The above example shows that 557,670 vertices were touched in the above traversal. Adapting this traversal slightly to use the store/except pattern makes it far more efficient, touching only 100 vertices) as shown below:

```
gremlin> s = [root] as Set
==>v[0]
gremlin> target = '99'; c = 0;
root.both().except(s).store(s).sideEffect{c++}.loop(4){it.object.id !=
target && it.loops <= 10}.has('id',target).path().iterate(); c
==>100
```

Subgraphing

Extracting a portion of a graph out of another graph by side-effecting out graph elements to another graph. Using a TinkerGraph as the host for the subgraph is best (as memory allows), since TinkerGraph will preserve element identifiers after the extractions.

```
// the goal is to extract the "knows" subgraph
gremlin> g.E.has('label','knows')
==>e[7][1-knows->2]
```

```
==>e[8][1-knows->4]

// this is the target subgraph
gremlin> sg = new TinkerGraph()
==>tinkergraph[vertices:0 edges:0]

// define a "Get Or Create" function for vertices...use ElementHelper to
copy properties
gremlin> def goc(v,g){nv=g.getVertex(v.id);if(nv==null)
{nv=g.addVertex(v.id,ElementHelper.getProperties(v))};nv}
==>true

// generate the subgraph by side-effecting graph elements into new graph
gremlin>
g.E.has('label','knows').sideEffect{sg.addEdge(it.id,goc(it.outV.next(),sg
),goc(it.inV.next(),sg),it.label,ElementHelper.getProperties(it))}.iterate
()
==>null
gremlin> sg.E
==>e[7][1-knows->2]
==>e[8][1-knows->4]
```

Using External Classes

Classes in external `jar` files can be referenced in the Gremlin REPL, making it possible to expand the capabilities of the REPL itself. Either copy the `jar` file to the `GREMLIN_HOME/lib` directory before starting the REPL so that the `jar` is on the Gremlin classpath or use Grape to help manage `jar` files and their related dependencies. The following example demonstrates how to use MongoDB from the Gremlin REPL:

```
gremlin> Grape.grab(group:'com.gmongo',module:'gmongo',version:'1.2')
==>null
gremlin> import com.gmongo.GMongo
==>import com.tinkerpop.gremlin.*
==>import com.tinkerpop.gremlin.java.*
==>import com.tinkerpop.gremlin.pipes.filter.*
...
==>import com.gmongo.GMongo
gremlin> mongo = new GMongo()
==>com.gmongo.GMongo@3717ee94
```

Writing To File

TinkerPop supports a number of different graph file formats, like GraphML, GML, and GraphSON, but sometimes a custom format or just a simple edge list is desireable. The following code shows how to open a file and side-effect out a comma-separated file of in and out vertices for each edge in the graph.

```
gremlin> new File("/tmp/edge-set.txt").withWriter{f -> g.E.sideEffect{f <<
"${it.outV.id.next()},${it.inV.id.next()}\r\n"}.iterate()}
==>null
```