NAME

Lintian::DepMap – Dependencies map/tree creator

SYNOPSIS

```
use Lintian::DepMap;

my $map = Lintian::DepMap->new;

# know about A:
$map->add('A');
# B depends on A:
$map->add('B', 'A');

# prints 'A':
print $map->selectable;

# indicate we are working on 'A' (optional):
$map->select('A');
# do 'A' ... work work work

# we are done with A:
$map->satisfy('A');
# prints 'B':
print $map->selectable;
```

DESCRIPTION

Lintian::DepMap is a simple dependencies map/tree creator and ''resolver''. It works by creating a tree based on the indicated dependencies and destroying it to resolve it.

Note: in the below documentation a `node` means a node name; no internal reference is ever returned and therefore never accepted as a parameter.

new()

Creates a new Lintian::DepMap object and returns a reference to it.

initialise()

Ensure, by reconstructing if necessary, the map's status is the initial. That is, partially or fully resolved maps can be restored to its original state by calling this method.

This can be useful when the same map will be used multiple times.

E.g.

```
$map->add('A');
$map->satisfy('A');
# prints nothing
print $map->selectable;
$map->initialise;
print $map->selectable;
```

add(node[, dependency[, dependency[, ...]]])

Adds the given `node` to the map marking any second or more parameter as its dependencies. E.g.

```
# A has no dependency:
$map->add('A');
# B depends on A:
$map->add('B', 'A');
```

addp(node[, prefix, dependency[, dependency[, ...]]])

Adds the given `node` to the map marking any third or more parameters, after prefixing them with `prefix`, as its dependencies. E.g.

```
# pA and pB have no dependency:
$map->add('pA');
$map->add('pA');
# C depends on pA and pB:
$map->addp('C', 'p', 'A', 'B');
```

satisfy(node)

Indicates that the given `node` has been satisfied/done.

The given `node` is no longer marked as being selected, if it was; all of its branches that have no other parent are now **selectable()** and all the references to `node` are deleted except the one from the **known()** list.

E.g.

```
# A has no dependencies:
$map->add('A');
# B depends on A:
$map->add('B', 'A');
# we work on A, and we are done:
$map->satisfy('A');
# B is now available:
$map->selectable('B');
```

Note: shall the requested node not exist this method **die()**s.

done(node)

Returns whether the given `node` has been satisfied/done.

E.g.

```
# A has no dependencies:
$map->add('A');
# we work on A, and we are done:
$map->satisfy('A');

print "A is done!"
    if ($map->done('A'));
```

unlink(node)

Removes all references to the given `node` except for the entry in the **known()** table.

IMPORTANT: since all references are deleted it is possible that a node that depended on `node` may become available even when it was not expected to.

IMPORTANT: this operation can **not** be reversed by the means of **initialise()**.

E.g.

```
$map->add('A');
# Prints A
print $map->selectable;
# we later notice we don't want A
$map->unlink('A');
# Prints nothing
print $map->selectable;
```

Note: shall the requested node not exist this method **die()**s.

select(node)

Marks the given `node` as selected to indicate that whatever it represents is being worked on. Note: this operation is not atomic.

E.g.

```
$map->add('A');
$map->add('B', 'A');
while($map->pending) {
    for my $node ($map->selectable) {
        $map->select($node);
        # work work work
        $map->satisfy($node);
    }
}
```

selectable([node])

If a `node` is specified returns TRUE if it can be **select()**ed.

Note: already **select()**ed nodes cannot be re-selected, i.e. if the given `node` has already been selected this function will return FALSE; or any selected item will be omitted from the returned array, in case no `node` is specified.

selected([node])

If a `node` is specified returns TRUE if it is has been selected, FALSE otherwise.

If no `node` is specified it returns an array with the name of all the nodes that have been **select()**ed but not yet satisfied.

E.g.

```
# We are going to work on A
$map->select('A');
# Returns true
$map->selected('A');
# Prints A
print $map->selected;
```

selectAll()

select()s all the **selectable()** nodes.

parents(node)

Return an array with the name of the parent nodes for the given `node`.

E.g.

```
$map->add('A');
$map->add('B', 'A');
# Prints 'A'
print $map->parents('B');
```

Note: shall the requested node not exist this method **die()**s.

ancestors(node)

Return an array with unique names of all ancestral nodes for the given `node`.

E.g.

```
$map->add('A');
$map->add('B', 'A');
# Prints 'A'
print $map->ancestors('B');
```

Note: shall the requested node not exist this method **die()**s.

non_unique_ancestors(node)

Return an array with non-unique names of all ancestral nodes for the given `node`.

E.g.

```
$map->add('A');
$map->add('B', 'A');
# Prints 'A'
print $map->ancestors('B');
```

Note: shall the requested node not exist this method **die**()s.

pending()

Return the number of nodes that can or have already been selected. E.g.

```
$map->add('B', 'A');
# prints 1:
print $map->pending;
$map->select('A');
# prints 1:
print $map->pending;
$map->satisfy('A');
# prints 1 ('B' is now available):
print $map->pending;
```

known()

Return an array containing the names of nodes that were added. E.g.

```
$map->add('B', 'A');
# prints 'B':
print $map->known;
$map->add('A');
# prints 'A' and 'B':
print $map->known;
```

known(NODE)

Returns a truth value if NODE is known or `undef` otherwise.

missing()

Return an array containing the names of nodes that were not added but that another node depended on it. E.g.

```
$map->add('B', 'A');
# prints 'A':
print $map->missing;
$map->add('A');
# prints nothing:
print $map->missing;
# this also works; A depends on 'Z':
$map->add('A', 'Z');
# but now this prints 'Z':
print $map->missing;
```

circular(['deep'])

Returns an array of nodes that have a circular dependency.

E.g.

```
$map->add('A', 'B');
$map->add('B', 'A');
# Prints A and B
print $map->circular;
```

Note: since recursive/deep circular dependencies detection is a bit more resource expensive it is not

the default.

```
$map->add('A', 'B');
$map->add('B', 'C');
$map->add('C', 'A');
# No deep/recursive scanning is performed, prints nothing
print $map->circular;
# deep scan, prints 'A, B, C'
print $map->circular('deep');
```

AUTHOR

Originally written by Raphael Geissert <atomo64@gmail.com> for Lintian.