

# Supplementary documentation on Relationship Manager constraints and Count() methods.

```
class RelationshipManagerConstrained:
"""
    The Constrained relationship manager is the recommended interface
    to relationship manager.

    You don't need to call EnforceRelationship() if you don't want to.

    The cardinality options are ("onetoone", "onetomany"). The default is
    no enforcement of cardinality. The result
    of breaking these rules is simple overwrite behaviour (no exceptions are raised).
    In other words, if you have a "onetotone" enforcement then try to break it by
    trying to create a one to many, then the new relationship will simply replace
    the old one, thus maintaining the constraint of one to one.

    The directionality options are ("directional", "bidirectional"). The default is
    "directional". "bidirectional" automatically adds a second relationship of
    the same relId in the reverse direction. Thus you get two relationships for every
    call to AddRelationship(). Similarly, when removing relationships,
    two relationships are removed for every call to RemoveRelationship().

    Note that the ability to use the backpointer feature of relationship manager
    i.e. calling FindObjectsPointingToMe() is not affected by the directionality
    setting. You can always get the backpointer, no matter what what the direction
    of the relationship is. Why bother with 'bidirectional' then? Well you may
    prefer to formalise you relationships and make things explicit, rather than
    relying on the magic powers of the relationship manager engine. Thus in a
    bidirectional situation you would not need to call for the backpointer,
    you would instead ask for the appropriate official forward pointer
    (two official forward pointers exist, remember).
"""

def CountRelationships(relId):
"""
    Returns a count of all the relationships matching the given relId
    in the relationship manager.

    Warning: There will be double the amount of relationships if you are using
    a "bidirectional" relationship i.e. For each AddRelationship() two
    relationships are added (both with the same relId). One is [from, to]
    and the other is [to, from].

    If you want to avoid this, then use a "directional" relationship. You
    can still get backpointers using the usual FindObjectsPointingToMe()
    method, even with simple "directional" relationships. See the main doco
    on why you would use "directional" relationships when simple "directional"
    give you all the facilities you want...
"""

def Count():
"""
    Returns a count of all the relationship entries in the relationship manager.

    Warning: There will be double the amount of relationships if you are using
    a "bidirectional" relationship i.e. For each AddRelationship() two
    relationships are added (both with the same relId). One is [from, to]
    and the other is [to, from].

    If you want to avoid this, then use a "directional" relationship. You
    can still get backpointers using the usual FindObjectsPointingToMe()
    method, even with simple "directional" relationships. See the main doco
    on why you would use "directional" relationships when simple "directional"
    give you all the facilities you want...

```

PTO. For interesting test case code...

# Test Cases proving you can use backpointer functionality with both directional and bidirectional relationships.

```
[TestFixture]
class TestCaseBidirectionality01:
    """
    The ability to use the backpointer feature of relationship manager
    i.e. calling FindObjectsPointingToMe() is not affected by the directionality
    setting. You can always get the backpointer, no matter what the direction
    of the relationship is. Why bother with 'bidirectional' then? Well you may
    prefer to formalise your relationships and make things explicit, rather than
    relying on the magic powers of the relationship manager engine. Thus in a
    bidirectional situation you would not need to call for the backpointer,
    you would instead ask for the appropriate official forward pointer
    (two official forward pointers exist, remember).

    """

    private _rm as RelationshipManagerConstrained

    [SetUp]
    def SetUp():
        self._rm = RelationshipManagerConstrained()

    [Test]
    def check_CanGetBackpointerOnDirectional():

        _rm.EnforceRelationship('r1', 'onetoone', 'directional')
        _rm.AddRelationship('a', 'b', 'r1')

        assert _rm.FindObjectPointingToMe('b', 'r1') == 'a'

    [Test]
    def check_CanGetBackpointerOnBiDirectional():

        _rm.EnforceRelationship('r1', 'onetoone', 'bidirectional')
        _rm.AddRelationship('a', 'b', 'r1')

        // Use magic backpointer abilities of RM
        assert _rm.FindObjectPointingToMe('b', 'r1') == 'a'

        // Or use the official backward pointer, generated by the 'bidirectional' option
        // (since it is really just a normal forward pointer, we use ...PointedToByMe not
        // the backpointer magic of ...PointingToMe)
        assert _rm.FindObjectPointedToByMe('b', 'r1') == 'a'
```