Defining Collection Methods

On This Page

When you create accessors and ivars using standard naming conventions, as described in Achieving Basic Key-Value Coding Compliance, the key-value coding protocol's default implementation can locate them in response to key-value coded messages. This is as true for collection objects representing to-many relationships as it is for other properties. However, if you implement the collection accessor methods instead of, or in addition to, the basic accessors for a collection property, you can:

- **Model to-many relationships with classes other than NSArray or NSSet.** When you implement the collection methods in your object, the default implementation of the key-value getter returns a proxy object that calls these methods in response to subsequent `NSArray` or `NSSet` messages that it receives. The underlying property object need not be an `NSArray` or `NSSet` itself, because the proxy object provides the expected behavior using your collection methods.
- **Achieve increased performance when mutating the contents of a to-many relationship.** Instead of repeatedly creating new collection objects with the basic setter in response to every change, the protocol's default implementation uses your collection methods to mutate the underlying property in place.
- **Provide key-value observing compliant access to the contents of your object's collection properties.** For more information about key-value observing, read *Key-Value Observing Programming Guide*.

You implement one of two categories of collection accessors, depending on whether you want the relationship to behave like an indexed, ordered collection (like an `NSArray` object) or an unordered, uniqued collection (like an `NSSet` object). In either case, you implement at least one set of methods to support read access to the property, and then add an additional set to enable mutation of the collection's contents.

> NOTE
>
> The key-value coding protocol does not declare the methods described in this section. Instead, the default implementation of the protocol provided by `NSObject` looks for these methods in your key-value coding compliant object, as described in Accessor Search Patterns, and uses them to handle key-value coded messages that are part of the protocol.

Accessing Indexed Collections

You add indexed accessor methods to provide a mechanism for counting, retrieving, adding, and replacing objects in an ordered relationship. The underlying object is often an instance of `NSArray` or `NSMutableArray`, but if you provide the collection accessors, you enable any object property for which you implement these methods to be manipulated as if it were an array.

Indexed Collection Getters

For a collection property that has no default getter, if you provide the following indexed collection getter methods, the default implementation of the protocol, in response to a `valueForKey:` message, returns a proxy object that behaves like an `NSArray`, but calls the following collection methods to do its work.

> NOTE
>
> In modern Objective-C, the compiler synthesizes a getter by default for each property, so the default implementation does *not* create a read only proxy that uses the methods in this section (note the accessor search order in Search Pattern for the Basic Getter). You can get around this by either not declaring a property (relying solely on an ivar), or by declaring a property as `@dynamic`, indicating that you plan to supply accessor behavior at runtime. Either way, the compiler will not supply a default getter, and the default implementation uses the methods below.

- `countOf<Key>`

  This method returns the number of objects in the to-many relationship as an `NSUInteger`, just like the `NSArray` primitive method `count`. In fact, when the underlying property is an `NSArray`, you use that method to provide a result.

  For example, for a to-many relationship representing a list of bank transactions and backed by an `NSArray` called `transactions`:

```
- (NSUInteger)countOfTransactions {
    return [self.transactions count];
}
```

- `objectIn<Key>AtIndex:` or `<key>AtIndexes:`

  The first returns the object at the specified index in the to-many relationship, while the second returns an array of objects at the indexes specified by the `NSIndexSet` parameter. These correspond to the `NSArray` methods `objectAtIndex:` and `objectsAtIndexes:`, respectively. You only need to implement one of these. The corresponding methods for the `transactions` array are:

```
- (id)objectInTransactionsAtIndex:(NSUInteger)index {
    return [self.transactions objectAtIndex:index];
}

- (NSArray *)transactionsAtIndexes:(NSIndexSet *)indexes {
    return [self.transactions objectsAtIndexes:indexes];
}
```

- `get<Key>:range:`

  This method is optional, but can result in improved performance. It returns the objects from the collection that fall within the specified range, and corresponds to the `NSArray` method `getObjects:range:`. The implementation for the transactions array is:

```
- (void)getTransactions:(Transaction * __unsafe_unretained *)buffer
               range:(NSRange)inRange {
    [self.transactions getObjects:buffer range:inRange];
}
```

Indexed Collection Mutators

Supporting a mutable to-many relationship with indexed accessors requires implementing a different group of methods. When you provide these setter methods, the default implementation, in response to the `mutableArrayValueForKey:` message, returns a proxy object that behaves like an `NSMutableArray` object, but uses your object's methods to do its work. This is generally more efficient than returning an `NSMutableArray` object directly. It also enables the contents of a to-many relationship to be key-value observing compliant (see *Key-Value Observing Programming Guide*).

In order to make your object key-value coding compliant for a mutable ordered to-many relationship, implement the following methods:

- `insertObject:in<Key>AtIndex:` or `insert<Key>:atIndexes:`

  The first receives the object to insert and an `NSUInteger` that specifies the index where it should be inserted. The second inserts an array of objects into the collection at the indices specified by the passed `NSIndexSet`. These are analogous to the `NSMutableArray` methods `insertObject:atIndex:` and `insertObjects:atIndexes:`. Only one of these methods is required.

  For a `transactions` object declared as an `NSMutableArray`:

```
- (void)insertObject:(Transaction *)transaction
  inTransactionsAtIndex:(NSUInteger)index {
    [self.transactions insertObject:transaction atIndex:index];
}

- (void)insertTransactions:(NSArray *)transactionArray
              atIndexes:(NSIndexSet *)indexes {
    [self.transactions insertObjects:transactionArray atIndexes:indexes];
}
```

- `removeObjectFrom<Key>AtIndex:` or `remove<Key>AtIndexes:`

  The first receives an `NSUInteger` value specifying the index of the object to be removed from the relationship. The second receives an `NSIndexSet` object specifying the indexes of the objects to be removed from the relationship. These methods correspond to the `NSMutableArray` methods `removeObjectAtIndex:` and `removeObjectsAtIndexes:` respectively. Only one of these methods is required.

For the `transactions` object:

```
- (void)removeObjectFromTransactionsAtIndex:(NSUInteger)index {

}

- (void)removeTransactionsAtIndexes:(NSIndexSet *)indexes {
    [self.transactions removeObjectsAtIndexes:indexes];
}
```

On This Page

- `replaceObjectIn<Key>AtIndex:withObject:` or `replace<Key>AtIndexes:with<Key>:`

  These replacement accessors provide the proxy object with a means to replace an object in the collection directly, without having to successively remove one object and insert another. They correspond to the `NSMutableArray` methods `replaceObjectAtIndex:withObject:` and `replaceObjectsAtIndexes:withObjects:`. You optionally provide these methods when profiling of your app reveals performance issues.

  For the `transactions` object:

```
- (void)replaceObjectInTransactionsAtIndex:(NSUInteger)index
                             withObject:(id)anObject {
    [self.transactions replaceObjectAtIndex:index
                              withObject:anObject];
}

- (void)replaceTransactionsAtIndexes:(NSIndexSet *)indexes
                    withTransactions:(NSArray *)transactionArray {
    [self.transactions replaceObjectsAtIndexes:indexes
                                  withObjects:transactionArray];
}
```

Accessing Unordered Collections

You add unordered collection accessor methods to provide a mechanism for accessing and mutating objects in an unordered relationship. Typically, this relationship is an instance of an `NSSet` or `NSMutableSet` object. However, when you implement these accessors, you enable any class to model the relationship and be manipulated using key-value coding just as if it were an instance of `NSSet`.

Unordered Collection Getters

When you provide the following collection getter methods to return the number of objects in the collection, iterate over the collection objects, and test if an object is already present in the collection, the default implementation of the protocol, in response to a `valueForKey:` message, returns a proxy object that behaves like an `NSSet`, but calls the following collection methods to do its work.

> NOTE
>
> In modern Objective-C, the compiler synthesizes a getter by default for each property, so the default implementation does *not* create a read only proxy that uses the methods in this section (note the accessor search order in Search Pattern for the Basic Getter). You can get around this by either not declaring a property (relying solely on an ivar), or by declaring a property as `@dynamic`, indicating that you plan to supply accessor behavior at runtime. Either way, the compiler will not supply a default getter, and the default implementation uses the methods below.

- `countOf<Key>`

  This required method returns the number of items in the relationship, corresponding to the `NSSet` method `count`. When the underlying object is an `NSSet`, you call on this method directly. For example, for an `NSSet` object called `employees` containing `Employee` objects:

```
- (NSUInteger)countOfEmployees {
    return [self.employees count];
}
```

- `enumeratorOf<Key>`

  This required method returns an `NSEnumerator` instance that is used to iterate over the items in the relationship. See Enumeration: Traversing a Collection's Elements in *Collections Programming Topics* for more i[illegible] `objectEnu`[illegible]. For the employees set.

```
- (NSEnumerator *)enumeratorOfEmployees {
    return [self.employees objectEnumerator];
}
```

- `memberOf<Key>:`.

  This method compares the object passed as a parameter with the contents of the collection and returns the matching object as a result, or `nil` if no matching object is found. If you implement comparisons manually, you typically use `isEqual:` to compare the objects. When the underlying object is an NSSet object, you can use the equivalent `member:` method:

```
- (Employee *)memberOfEmployees:(Employee *)anObject {
    return [self.employees member:anObject];
}
```

Unordered Collection Mutators

Supporting a mutable to-many relationship with unordered accessors requires implementing additional methods. Implement the mutable unordered accessors to allow your object to supply an unordered set proxy object in response to the `mutableSetValueForKey:` method. Implementing these accessors is much more efficient than relying on an accessor that returns a mutable object directly for making changes to the data in the relationship. It also makes your class key-value observing compliant for the collected objects (see *Key-Value Observing Programming Guide*).

In order to be key-value coding complaint for a mutable unordered to-many relationship implement the following methods:

- `add<Key>Object:` or `add<Key>:`

  These methods add a single item or a set of items to the relationship. When adding a set of items to the relationship, ensure that an equivalent object is not already present in the relationship. These are analogous to the `NSMutableSet` methods `addObject:` and `unionSet:`. Only one of these methods is required. For the `employees` set:

```
- (void)addEmployeesObject:(Employee *)anObject {
    [self.employees addObject:anObject];
}

- (void)addEmployees:(NSSet *)manyObjects {
    [self.employees unionSet:manyObjects];
}
```

- `remove<Key>Object:` or `remove<Key>:`

  These methods remove a single item or a set of items from the relationship. They are analogous to the `NSMutableSet` methods `removeObject:` and `minusSet:`. Only one of these methods is required. For example:

```
- (void)removeEmployeesObject:(Employee *)anObject {
    [self.employees removeObject:anObject];
}

- (void)removeEmployees:(NSSet *)manyObjects {
    [self.employees minusSet:manyObjects];
}
```

- `intersect<Key>:`

  This method, which receives an `NSSet` parameter, removes from the relationship all the objects that aren't common to both the input set and the collection set. This is the equivalent of the `NSMutableSet` method `intersectSet:`. You optionally implement this method when profiling indicates performance issues surrounding updates to the collection content. For example:

```
- (void)intersectEmployees:(NSSet *)otherObjects {
    return [self.employees intersectSet:otherObjects];
}
```

On This Page