

# Contents

- [Persisting and Retrieving Objects](#)
    - [Creating a New Object](#)
      - [Transfer.new\(class\)](#)
    - [Object Retrieval](#)
      - [Transfer.get\(class, key\)](#)
      - [Transfer.readByProperty\(class, property, value\)](#)
      - [Transfer.readByPropertyMap\(class, propertyMap\)](#)
      - [Transfer.readByQuery\(class, query\)](#)
    - [Persisting Objects](#)
      - [Transfer.create\(transferObject, \[useTransaction\]\)](#)
      - [Transfer.cascadeCreate\(transferObject, \[depth\], \[useTransaction\]\)](#)
      - [Transfer.update\(transferObject, \[useTransaction\]\)](#)
      - [Transfer.cascadeUpdate\(transferObject, \[depth\], \[useTransaction\]\)](#)
      - [Transfer.save\(transferObject, \[useTransaction\]\)](#)
      - [Transfer.cascadeSave\(transferObject, \[depth\], \[useTransaction\]\)](#)
    - [Deleting Objects](#)
      - [Transfer.delete\(transferObject, \[useTransaction\]\)](#)
      - [Transfer.cascadeDelete\(transferObject, \[depth\], \[useTransaction\]\)](#)
- 

## Persisting and Retrieving Objects

### Creating a New Object

**Transfer.new(class)**

[\(API\)](#)

This methods creates a new TransferObject, that has a default primary key. It has not been created in the database and is not stored in the persistent scope.

Note that the class name is case sensitive.

### Object Retrieval

**Transfer.get(class, key)**

[\(API\)](#)

This will retrieve a TransferObject from the database and place it in the cache that the object is configured for, for future retrieval. More details on caching, can be found at [Managing the Cache](#)

If a object of the given key does not exist, it will return an new, empty instance of the class.

Note that the class name is case sensitive.

### **Transfer.readByProperty(class, property, value)**

[\(API\)](#)

Retrieves a TransferObject of a given class by a property with a unique value.

If the record does not exist a new, empty instance is returned.

If the property value results in more than one result of the given class, an exception will be thrown.

### **Transfer.readByPropertyMap(class, propertyMap)**

[\(API\)](#)

Retrieves a TransferObject of a given class by a struct of values, where the struct key is the object property, and the value is the value to filter by.

If the record does not exist a new, empty instance is returned.

If the property value results in more than one result of the given class, an exception will be thrown.

### **Transfer.readByQuery(class, query)**

[\(API\)](#)

Retrieves a TransferObject of a given class by result of the TQL Query object that is passed to it.

TQL Queries that are used for readByQuery() operations can either start with a FROM statement, or can return a single columned result, that is the value of the *id* you wish to get of the given class (e.g. `Select Post.IDPost from...`).

It should be noted that TQL Query Objects are reused after they are executed, so they will lose their state after the readByQuery() operation.

For information on TQL and the TQL Query object, see [Transfer Query Language](#).

If the record does not exist a new, empty instance is returned.

If the TQL results in more than one result of the given class, an exception will be thrown.

## **Persisting Objects**

## **Transfer.create(transferObject, [useTransaction])**

([API](#))

This method takes an object created by the method `new()` and inserts it into the database.

If in the configuration the ID element has a [`@generate='true'`] value, Transfer will generate the primary key.

If [`@generate='false'`], or there is no 'generate' attribute, the database will attempt to retrieve the primary key value from the database.

If the ID is `set()` before the TransferObject is inserted, this will override both of these options, and Transfer will attempt to insert the data into the database under the set ID.

More details can be found in [Primary Key Management](#)

This also places this object inside the configured persistence scope of the class.

If this object is referenced as a child of a onetomany collection, it will be added to its set parents collection when it is created.

Any child objects that are added to a manytomany collection on this TransferObject has its links committed to the database.

If the object has already been created, an exception will be thrown.

If 'useTransaction' is set to false, Transfer won't use it's internal `cftransaction` blocks. Defaults to true. This option has been deprecated, in favour of the [Transfer Transactions](#), however can still be applicable in some scenarios.

## **Transfer.cascadeCreate(transferObject, [depth], [useTransaction])**

([API](#))

This method recursively cascades through *all* of the relationships the transferObject has, including both parents and children, and calls [transfer.create\(\)](#) on each TransferObject it finds.

The depth argument tells the cascade function how many levels deep it should travel into relationships. By default, there is no limit.

If 'useTransaction' is set to false, Transfer won't use it's internal `cftransaction` blocks. Defaults to true. This option has been deprecated, in favour of the [Transfer Transactions](#), however can still be applicable in some scenarios.

## **Transfer.update(transferObject, [useTransaction])**

([API](#))

This method updates the details of a TransferObject within the database, if the details of the object have been changed since it was last committed to the database.

If this TransferObject is referenced as a child of a onetomany collection that is in a cache, if the parent of this object has changed, it will both be committed to the database, and the TransferObject will be moved from the collection of one parent object to another.

If changes have been made to a collection of type manytomany on this TransferObject, children either removed or added, these changes are committed to the database at this stage.

If 'useTransaction' is set to false, Transfer won't use it's internal cftransaction blocks. Defaults to true. This option has been deprecated, in favour of the [Transfer Transactions](#), however can still be applicable in some scenarios.

### **Transfer.cascadeUpdate(transferObject, [depth], [useTransaction])**

[\(API\)](#)

This method recursively cascades through *all* of the relationships the transferObject has, including both parents and children, and calls [transfer.update\(\)](#) on each TransferObject it finds.

The depth argument tells the cascade function how many levels deep it should travel into relationships. By default, there is no limit.

If 'useTransaction' is set to false, Transfer won't use it's internal cftransaction blocks. Defaults to true. This option has been deprecated, in favour of the [Transfer Transactions](#), however can still be applicable in some scenarios.

### **Transfer.save(transferObject, [useTransaction])**

[\(API\)](#)

If this TransferObject has yet to be created, it runs the create(transferObject) function, otherwise it runs the update(transferObject) function

If 'useTransaction' is set to false, Transfer won't use it's internal cftransaction blocks. Defaults to true. This option has been deprecated, in favour of the [Transfer Transactions](#), however can still be applicable in some scenarios.

### **Transfer.cascadeSave(transferObject, [depth], [useTransaction])**

[\(API\)](#)

This method recursively cascades through *all* of the relationships the transferObject has, including both parents and children, and calls [transfer.save\(\)](#) on each TransferObject it finds.

The depth argument tells the cascade function how many levels deep it should travel into relationships. By default, there is no limit.

If 'useTransaction' is set to false, Transfer won't use it's internal cftransaction blocks. Defaults to true. This option has been deprecated, in favour of the [Transfer Transactions](#), however can still be applicable in some scenarios.

## Deleting Objects

### **Transfer.delete(transferObject, [useTransaction])**

([API](#))

This will delete a TransferObject from the database and the persistence scope if it has been inserted into the database.

If this TransferObject has a onetomany collection, it will set all its children to having no parent, and break a foreign key constraint, if it exists. This may cause an error, depending on your database model.

If this TransferObject is a member of a onetomany collection, or a manytomany collection that is cached, it will be removed.

If 'useTransaction' is set to false, Transfer won't use it's internal cftransaction blocks. Defaults to true. This option has been deprecated, in favour of the [Transfer Transactions](#), however can still be applicable in some scenarios.

### **Transfer.cascadeDelete(transferObject, [depth], [useTransaction])**

([API](#))

This method recursively cascades through *all* of the relationships the transferObject has, including both parents and children, and calls [transfer.delete\(\)](#) on each TransferObject it finds.

The depth argument tells the cascade function how many levels deep it should travel into relationships. By default, there is no limit.

If 'useTransaction' is set to false, Transfer won't use it's internal cftransaction blocks. Defaults to true. This option has been deprecated, in favour of the [Transfer Transactions](#), however can still be applicable in some scenarios.