

*Coding Java Applications on  
Cache-Forward Architecture  
White Paper*

## *Table of Contents*

<i>Introduction .....</i>	<i>3</i>
<i>Getting Started with the ObjectStore Java Interface .....</i>	<i>5</i>
<i>Creating and Joining Sessions .....</i>	<i>5</i>
<i>Creating and Opening Databases .....</i>	<i>6</i>
<i>Creating Transactions.....</i>	<i>7</i>
<i>Storing Objects.....</i>	<i>8</i>
<i>Creating Entry Points to Databases.....</i>	<i>9</i>
<i>Creating Collections of Objects .....</i>	<i>9</i>
<i>Retrieving Entry Points to Databases.....</i>	<i>10</i>
<i>Using Iterators to Navigate Collections.....</i>	<i>10</i>
<i>Querying Collections .....</i>	<i>11</i>
<i>Optimizing Queries .....</i>	<i>12</i>
<i>Ending Transactions.....</i>	<i>13</i>
<i>Conclusion.....</i>	<i>14</i>
<i>Glossary.....</i>	<i>15</i>
<i>About Progress Real Time Division.....</i>	<i>18</i>

## *Introduction*

Progress's Cache-Forward™ Architecture (CFA) can help you achieve the performance required by applications that:

Manage data for rich Java® object models

Cache enterprise data in the middle tier

Build real time, high-performance event processing systems, such as financial trading systems, fraud detection, or Radio Frequency Identification

When you are writing applications that take advantage of CFA, CFA's single-level storage model simplifies the way you approach stored data because:

Data you operate on is in memory.

There is no mapping from one data format to another.

You can work with object models that are as complex as you need them to be.

The interface for writing applications is standard Java.

There is a tight integration between CFA and Java. This lets developers who understand Java easily learn how to write applications that harness CFA. Writing CFA applications is just like programming with standard Java. With CFA, there is no separate data definition language or data manipulation language – the programming language is the interface.

CFA is the foundation for Progress® ObjectStore® Enterprise, Progress® Apama® EventStore™, and Progress® ObjectStore® PSE Pro®. ObjectStore Enterprise (typically referred to as ObjectStore) is a multiuser, enterprise-class, object database that stores objects in their native format. Progress Apama EventStore is a real-time, Event Stream Processing (ESP) product that manages data for event-driven applications in real time. ObjectStore PSE Pro is a small-footprint object database that is a replacement for file-based storage.

This white paper provides an overview of the ObjectStore Java API, which is common to ObjectStore and PSE Pro. The code fragments are part of a complete

application that you can download at

**[http://www.progress.com/realtime/publications/coding\\_on\\_cfa](http://www.progress.com/realtime/publications/coding_on_cfa)**.

This document is for potential customers who are experienced Java developers or development managers. No knowledge of CFA or CFA products is assumed. New CFA developers will also find this information useful.

For an overview of the CFA C++ API see *Coding C++ Applications on Cache-Forward Architecture*. You can download these papers at

**[http://www.progress.com/realtime/publications/coding\\_on\\_cfa](http://www.progress.com/realtime/publications/coding_on_cfa)**.

For a detailed description of CFA, see *Introducing Cache-Forward Architecture* at

**<http://www.progress.com/realtime/products/objectstore/index.ssp>**.

## *Getting Started with the ObjectStore Java Interface*

The ObjectStore Java interface is a library of classes that let you write applications that take advantage of CFA's features. You can use a variety of JVM implementations, together with the ObjectStore Java interface.

The ObjectStore Java API preserves the automatic storage management semantics of Java. Objects become persistent when they are referenced by other persistent objects. This is called persistence by reachability or *transitive persistence*. The application defines persistent roots and, when it commits a transaction, ObjectStore finds all objects reachable from persistent roots and stores them in the database.

You can store any type of value in persistent or transient memory. If you can describe it with Java, you can store it. CFA lets you use the full power of Java to define, create, and manipulate objects. This means full support for the complete language, including inheritance, polymorphism, encapsulation, and standard libraries.

Before you can access persistent memory, you must set the stage by performing a few operations:

- ⇒ Create and join a session
- ⇒ Create or open a database
- ⇒ Start a transaction
- ⇒ Retrieve or create a database root

The database features that let your applications scale to handle the ever increasing quantity and complexity of objects include collections and queries.

## *Creating and Joining Sessions*

A *session* allows the use of the ObjectStore Java API. ObjectStore uses the abstract `com.odi.Session` class to represent sessions. Your application must create a session before it can use any of the ObjectStore API.

A session creates a context in which you can create a transaction, access a database, and manipulate persistent objects. A session consists of a set of persistent objects and a set of ObjectStore API objects such as a Transaction, Databases, and Segments. In a single Java VM process, ObjectStore allows concurrent sessions.

Separate Java virtual machines can each run their own session at the same time. In addition, each Java virtual machine can run multiple sessions at the same time. Any number of Java threads can participate in the same session. A thread must join a session to be able to access and manipulate persistent objects.

To create a session, you call the static `create()` method on the `Session` class. To join a thread to a session, you call the `Session.join()` method. For example:

```
Session session = Session.create(null, null);  
session.join();
```

The benefit of a session is apparent when you want to have more than one session. Two sessions in the same Java virtual machine allow you to perform two distinct activities that involve ObjectStore. Each session has an isolated view of the database. If you want to have two or more independent transactions at the same time, you can use two or more sessions. Concurrent sessions can be accessing the same database or different databases.

## *Creating and Opening Databases*

You create databases in which to store your objects. The Database class is an abstract class that represents a database. When you create a database,

- ⇒ There must be an active session.
- ⇒ No transaction can be in progress in that active session.

Databases are cross-platform compatible. You can create databases on any supported platform and access them from any supported platform. To create a database, call the static create() method on the Database class. For example:

```
db = Database.create(dbpath, ALL_READ | ALL_WRITE);
```

This example creates an instance of Database and stores a reference to the instance in the variable named db. The Database.create() method takes two parameters. The first parameter specifies the pathname of the database file. The path specifies a relative name or a fully qualified name of a file that is one of the following:

- ⇒ Local
- ⇒ In a mounted directory
- ⇒ In an unmounted remote directory - in this case the file must be identified by a pathname that specifies a remote host

An ObjectStore server must be available to the directory that contains the specified file.

The second parameter specifies the access mode for the database.

For each database you create, ObjectStore creates an instance of Database to represent your database. Each database is associated with one instance of Database. Consequently, you can use the == operator to determine whether two Database objects in the same session represent the same database.

When you create a database, ObjectStore normally creates two segments:

- ⇒ The schema segment, which ObjectStore uses to hold schema information about the objects stored in the database and the database roots. The schema segment is not directly accessible by your application.
- ⇒ The default segment, which consists of clusters that ObjectStore uses when it is allocating persistent storage. A cluster is the basic unit of allocation in an ObjectStore database. Clusters are organized logically into segments.

You can create segments and clusters as needed, and specify any of these for allocation. Typically, you are concerned with clusters when you want to store data near each other or far away from each other. You are concerned with segments when you want to set the logical characteristics (for example, access control) of a group of clusters.

More than one database can be open in a single session.

The following code, from the Book.java file in the sample application, obtains the default segment, creates a cluster in the default segment, creates an OSTreeSet collection that contains Book objects, and stores the collection in the cluster just created. The getISBN() argument creates an index based on the Book object's ISBN number. See

Optimizing Queries on page 12 for information about indexes.

```
Segment segment = db.getDefaultSegment();
Cluster cluster = segment.createCluster();
extent = new OSTreeSet(cluster, Book.class, "getISBN()");
```

## Creating Transactions

A program must start a transaction before it accesses persistent data. While a transaction is in progress, a program can read and write persistent objects. At any time, a program can commit or abort a transaction to end it. When a program commits a transaction, the server makes permanent in the database any changes to persistent data made during the transaction. The server also makes any updates visible to other processes. When a program aborts a transaction, the server undoes or rolls back any changes made to persistent data during that transaction.

As you can see, transactions do two things:

- ⇒ They identify code sections whose effects can be undone.
- ⇒ They identify functional program areas that are isolated from changes made by other processes (clients). The data seen by a transaction remains consistent throughout that transaction and does not change if another update transaction commits during this transaction. From the point of view of other processes, these functional sections execute either all at once or not at all. That is, other processes do not see intermediate results.

Only one transaction can be running in a session at a time.

In your code, you place calls to the ObjectStore API to mark the beginnings and ends of transactions. Depending on how you end the transaction, additional access to previously stored objects might be possible.

ObjectStore provides the `com.odi.Transaction` class to represent a transaction. To start a transaction, you call the `begin()` method on the `Transaction` class. This returns an instance of `Transaction` and you can assign it to a variable. For example:

```
Transaction t = Transaction.begin(UPDATE);
```

This example returns a `Transaction` object that represents the transaction just started. The returned object is stored in `t`. The argument to the `begin()` method indicates the transaction type. This is an update transaction, which means that the application can modify database contents. To modify persistent objects, you must specify the transaction type to be `ObjectStore.UPDATE`.

The type of the transaction can be `ObjectStore.UPDATE` or `ObjectStore.READONLY`. The major difference between the two transaction types is that when you start a transaction for `READONLY`, ObjectStore performs additional checks during the transaction and when you commit the transaction. These checks ensure that changes are not saved in the database if they were made in a read-only transaction. There is no difference in performance between a read-only transaction and an update transaction that only reads objects.

Separate transactions that access the same database compete with one another for locks on the objects that they access. This can cause one transaction to wait for another to release its locks. Alternatively, it can cause a transaction deadlock situation in which two or more transactions wait for each other. This forces one transaction to abort.

Two transactions can never update the same object at the same time. However, two transactions can both open the same database for update at the same time and they can make updates concurrently to different parts of the database.

For details about the ObjectStore locking model, see the *Introducing Cache-Forward Architecture* white paper. Numerous examples of using transactions are in the sample programs. You can download both the white paper and the sample programs at <http://www.progress.com/realtime/products/objectstore/index.ssp>.

## Storing Objects

There are several ways to store an object:

- ⇒ An application assigns a transient object to a database root. ObjectStore immediately stores the object in the default cluster of the default segment in the database. When the transaction commits, if there are any transient objects that are reachable from the object assigned to the root, ObjectStore stores them in the default segment and cluster.
- ⇒ A transient object is reachable from a persistent object. When the transaction commits, ObjectStore stores the reachable object in the same database, segment and cluster as the persistent object.
- ⇒ An application invokes the ObjectStore.migrate()\_method on a transient object and specifies a particular database, segment, or cluster. ObjectStore immediately stores the migrated object in the specified location. When the transaction commits, ObjectStore also stores in the same location any transient objects that are reachable from the migrated object.

To store an object in a database, the object must be *persistence capable*. The term persistence capable refers to the capacity of an object to be stored in a database. If you can store the instances of a class in a database, the class is a persistence-capable class and the instances are persistence-capable objects.

The definition of a persistence-capable class includes specific annotations required by the ObjectStore API. To make a class persistence capable, you compile the class definition and then you run the ObjectStore-provided class file postprocessor on the compiled class. The postprocessor adds the annotations that make the classes persistence capable. In unusual circumstances, you might choose to add the annotations to the Java source file manually.

When you write an ObjectStore program, you write it as though classes are persistence capable. However, a program cannot store objects persistently until you run the postprocessor.

You must explicitly postprocess or manually annotate each class that you want to be persistence capable. The capacity for an object to be stored in a database is not inherited when you subclass a persistence-capable class.

Some Java-supplied classes are persistence capable. Other Java-supplied classes are not and cannot be made persistence capable. A third category of classes can be made persistence capable, but there are important issues to consider when you do so. See the *ObjectStore Java API User Guide* for a discussion of these issues and to determine which Java-supplied classes can be persistence capable.

The term *persistent object* refers to a representation in the Java virtual machine of an object that is stored in a database. After an application retrieves an object from the database, the application works with the persistent object in the Java environment.

When two sessions are accessing the same object in the database, there are two distinct persistent objects. Each session has its own persistent object, which is a copy of the object in the database. At least initially, these two persistent objects have the same content. Each session has its own set of persistent objects and API objects. In most circumstances, the threads of session A are not allowed to operate on the objects of session B.

## *Creating Entry Points to Databases*

A database root is a reference to an individual object in a database. Each database must have one or more database roots. Each database root is an entry to a database in that you use the root to navigate to objects in the database.

When you create a database root, you give it a name and you assign an object to it. The database root refers to that object, and your application can use the root name to access that object.

You create a database root inside a transaction with a call to the `Database.createRoot()` method on the database in which you want to create the root. For example:

```
db.createRoot("books", extent);
```

This example creates a root in the database referred to by `db`. In this example, the name of the root is `books`, and the object that the root refers to is a collection of `Book` objects called `extent`. The name you specify for the root must be unique in the database. The object that you specify to be referred to by the root can be either transient and persistence capable, or it can already have been stored in the database. If it has not yet been stored in the database, `ObjectStore` immediately stores it.

When the transaction commits, `ObjectStore` stores in the database referred to by `db` any objects that are reachable from `extent`, if they are not already in the database. If `extent` or any object it references refers to any transient objects that are not persistence capable and you try to commit the transaction, `ObjectStore` signals an exception.

In general, you should not create a root for each object you want to store in a database. You must create at least one root. If a database has only one root, you must be able to reach all other objects in that database from that root. A persistent Java object can simply reference another persistent Java object. You can get by with a single root, but you might find it convenient to have additional roots. Each root refers to exactly one object. That object can refer to other objects, which can refer to yet more objects, and so on. More than one root can refer to the same object.

One persistent object can reference another persistent object. You can store arbitrary graphs of persistent objects, including lists, trees, cycles, or any other graph your application requires. You can navigate from object to object by following references just as you do in transient structures. When you follow a reference to a persistent object that has not been seen before, that object is fetched from the database.

## *Creating Collections of Objects*

A *collection* is an object that groups together other objects. It provides an effective means of storing and manipulating groups of objects and supports operations for inserting, removing, and retrieving elements.

Collections form the basis of the `ObjectStore` query facility, which allows you to select those elements of a collection that satisfy a specified condition. However, some collections can be queried and others cannot. Therefore, before you create a collection and store it in a database, you should consider how you plan to use a collection. When you know what you need, you can select the best persistent collection representation for your application.

`ObjectStore` provides a number of utility collections interfaces and classes in the `com.odi.util` package. In addition, `ObjectStore` provides a query facility in the `com.odi.util.query` package. For example, the following fragment of the `Customer` class definition declares that an `OSHashSet` collection contains that customer's orders, and that a static `OSTreeMapString` collection contains all customer objects.

```

public class Customer {
    private String name;
    /* Orders made by this Customer are in an OSHashSet collection. */
    private OSHashSet orders;
    /* A persistent map contains all Customer objects. */
    private static OSTreeMapString map;
    ...
}

```

An `OSHashSet` is an unordered collection that does not allow duplicates. If you try to insert a value into an `OSHashSet` and the set already contains that value, the set remains unchanged. `OSHashSet` implements the `java.util.Set` interface. As its name implies, the internal representation of an `OSHashSet` collection is a hash table. Because `OSHashSet` indirectly implements `java.util.Collection`, you can query an `OSHashSet` collection.

The `OSTreeMapxxx` classes are designed for large persistent aggregations. These classes allow you to iterate over the collection without fetching any objects from the database except those that are explicitly returned to you. `ObjectStore` does not create hollow objects to represent the elements until they are fetched, thus reducing Java heap overhead when a subset of `OSTreeMapxxx` is accessed.

## *Retrieving Entry Points to Databases*

When you retrieve a root object, you obtain a reference to the object that is the value of the root. For example, the `Book` class defines an `OSTreeSet` object as the value of a root named `books`. When you get the value of `books` by using the `Database.getRoot()` method, you receive a reference to the `OSTreeSet` object as follows:

```
extent = (OSTreeSet)db.getRoot("books");
```

Note that `ObjectStore` does not fetch the entire contents of the `OSTreeSet` until you actually need it. This means that `ObjectStore` has not yet fetched the contents of the elements in the collection. `ObjectStore` fetches the data for an element in a collection only when you try to access its contents. You can obtain a reference to any object in a collection. In some cases, you might want to force `ObjectStore` to fetch an entire graph of connected objects even though the application does not explicitly refer to each object in the graph. Use the `ObjectStore.deepFetch()` method to do this. To obtain a list of the roots in a database, call the `getRoots()` method on the database.

In summary, you can access persistent objects in two ways:

- ⇒ As database roots
- ⇒ By reachability from other persistent objects

## *Using Iterators to Navigate Collections*

The `Iterator` and `ListIterator` interfaces help you navigate within a collection. An *iterator* is an instance of the `java.util.Iterator` or `java.util.ListIterator` interface. It designates a position in a collection. You can use iterators to traverse collections as well as to add and remove elements from collections.

The `ListIterator` interface extends the `Iterator` interface. A class that supports traversal by `ListIterator` must also implement `List`. The additional methods that `ListIterator` provides allow you to

- ⇒ Insert objects relative to the current position of the iterator
- ⇒ Traverse the list in reverse as well as forward

- ⇒ Replace an element in the underlying list
- ⇒ Retrieve the index of an element

While you are iterating through a collection, you can use the `Iterator` and `ListIterator` interface methods to modify that collection. When a thread is iterating over a collection, that thread and cooperating threads can modify the object returned by the iterator.

For example, the `ReadCustomer` class iterates through the collection of customers as follows:

```
Iterator customerIterator = Customer.iterator();
while (customerIterator.hasNext()) {
    aCustomer = (Customer)customerIterator.next();
    System.out.println(aCustomer);
}
```

## *Querying Collections*

The `com.odi.util.query.Query` class provides a mechanism for querying collection objects that implement the `java.util.Collection` interface. A query applies an expression that evaluates to a boolean result to all elements in a collection. The query returns a subset collection that contains all elements for which the expression is true. You can query the following classes that implement the `Collection` interface:

- ⇒ `OSHashBag`
- ⇒ `OSHashSet`
- ⇒ `OSTreeSet`
- ⇒ `OSVector`
- ⇒ `OSVectorList`

To create a query, run the `com.odi.util.query.Query` constructor and pass in a `Class` object and a query string. Following is the constructor:

```
public Query(Class elementType, String queryExpression)
```

There is also a constructor that allows you to specify a `FreeVariables` map.

The `elementType` class or interface provides the context in which the query facility interprets `queryExpression`. This must be a publicly accessible class or interface. When your application calls the `Query.select()` or `Query.pick()` method to execute the query against a particular collection, every element of that collection must be an instance of (in the sense of `instanceof`) the `elementType` that was specified when the query was created. Any element of the collection that is not an instance of `elementType` is not returned in the query result, even if it would evaluate to true for the query expression.

The `queryExpression` is an expression with a boolean result that the query facility evaluates for each element of the collection. The `queryExpression` operands can be literals and names.

Literals can be of any of the Java primitive types, including the special values `true`, `false`, and `null`. Because the query expression is a `String`, you must enclose any embedded strings in escaped quotation marks, like `"this\"`.

Names can consist of a single identifier or they can consist of a sequence of identifiers separated by periods. Names can be either free variables or member names (field or method names). You must explicitly specify free variables in the `freeVariables` argument of the three-argument `Query` constructor. Any name that is not a free variable is interpreted as a member name.

Member accesses are interpreted as accessing public members, including static members of an object of class/interface `elementType`, if possible. This interpretation works as though there were an implicit `this` argument of `elementType` at the root of the name expression. Any member access that cannot be interpreted as a member access on `elementType` is interpreted as a static access. Static accesses are resolved as if the package containing `elementType` were imported.

Queries can contain methods that take arguments. The arguments can be literals or free variables.

For example, the `QueryBook` class defines a query that returns the set of books that each cost more than \$15.00:

```
java.util.Set resultsCollection1 = Book.query("getCost() > 15.00");
```

The code for iterating through `resultsCollection1` is as follows :

```
Iterator resultsIterator1 = resultsCollection1.iterator();
System.out.println("\nHere are books that cost more than $15.00:");
while (resultsIterator1.hasNext()) {
    Book aBook = (Book)resultsIterator1.next();
    System.out.println(aBook);
}
```

The query expression can refer to classes without specifying a package name. `ObjectStore` treats the query expression as if it were defined in a file in another package that has imported the package of the `Class` object that was passed to the `Query` constructor. This default package matters only for class names, though, not for member access. Only public classes and members are accessible within the query.

When you create a query, you do not bind it to a particular collection. You can create a query, run it once, and throw it away. Alternatively, you can reuse a query multiple times against the same collection, perhaps with different bindings for free variables, or against different collections.

If the syntax of your query is wrong, `QueryException` is thrown at the point at which you create the query. You need not wait for the application to optimize or to execute the query. However, the query facility cannot detect incorrect free variable bindings until you specify them when you execute the query on a collection.

## *Optimizing Queries*

When you want to run a query on a particularly large collection, it is useful to build indexes on the collection to accelerate query processing. An index provides a reverse mapping from a field value, or from the value returned by a method when it is called, to all elements that have the value. A query that refers to an indexed member executes faster because it is not necessary to examine each object in the collection to determine the elements that match the query expression. Also, `ObjectStore` does not need to fetch every element into memory.

When you add an index to a collection, `ObjectStore` examines every element of the collection to determine the value of the indexed field or method. After you build the index, you can run queries against the collection without reexamining the elements to determine the values of any indexed members. The query examines the index instead of the collection.

A query can include both indexed fields and methods, and nonindexed fields and methods. `ObjectStore` evaluates the indexed fields and methods first and establishes a preliminary result set. `ObjectStore` then applies the nonindexed fields and methods to the elements in the preliminary result set.

You can add indexes to any collection that implements the `com.odi.util.IndexedCollection` interface, directly or indirectly. Note that the `IndexedCollection` interface extends the `Collection` interface. Call the `com.odi.util.IndexedCollection.addIndex()` method to create an index. For example:

The following code, from the `Book.java` file in the sample application, shows the creation of several indexes and the addition of those indexes to the collection of books.

```
// Create a collection of Book objects. The last argument specifies the creation
// of an index based on the book's ISBN number.
extent = new OSTreeSet(cluster, Book.class, "getISBN()");
// Add an index to the Book collection on the author:
Cluster authorIndexCluster = segment.createCluster();
extent.addIndex(Book.class, "getAuthor()", true, true, authorIndexCluster);
// Add an index to the Book collection on the cost:
Cluster costIndexCluster = segment.createCluster();
Extent.addIndex(Book.class, "getCost()", true, true, costIndexCluster);
```

## *Ending Transactions*

When you commit a transaction, `ObjectStore`

- ⇒ Saves and commits any modifications in the database
- ⇒ Checks for transient objects that are referred to by persistent objects

If there are such objects, `ObjectStore` stores them in the database if they are persistence capable. This is called transitive persistence. All reachable persistence-capable objects become persistent through transitive persistence.

If the modifications contain references to objects that are not persistence capable, `ObjectStore` signals `AbortException`.

- ⇒ Sets the state of persistent objects after the transaction.

If objects were stored in the database for the first time during this transaction, the copies of these objects in your Java program are included in the group of persistent objects.

By default, persistent objects are stale after a transaction. Stale objects, as well as references to stale objects, are no longer valid and cannot be reused. When you do not want to make your persistent objects stale, you call the `commit()` method and specify a retain state that applies to the persistent objects in the transaction you are committing. The possible retain states are `hollow`, `stale`, `read-only`, `update`, and `transient`. For example:

```
tx = Transaction.begin(ObjectStore.UPDATE);
init(db); // Initialize a database and retrieve some objects.
tx.commit(ObjectStore.RETAIN_HOLLOW);
```

The `ObjectStore.RETAIN_HOLLOW` constant indicates that the persistent objects and references to those objects are hollow when the transaction ends. Between transactions, the contents of hollow objects are not accessible. This is because the contents are in a database and database contents are not accessible when a transaction is not in progress. When a new transaction begins, the hollow persistent objects become active persistent objects. You can use the objects and the references to those objects without further queries or navigation.

Persistent objects that are not referenced directly or indirectly by the application, and are not modified in the current transaction, can be reclaimed by the Java garbage collector. Such objects are refetched from the database if needed again.

## *Conclusion*

When you are writing applications that take advantage of CFA, ObjectStore's single-level storage model simplifies the way you approach stored data because:

- ⇒ Data you operate on is in memory.
- ⇒ There is no mapping from one data format to another.
- ⇒ You can work with object models that are as complex as you need them to be.
- ⇒ The interface for writing applications is standard Java.

The basic ObjectStore operations when coding a CFA Java application are

- ⇒ Creating and joining a session
- ⇒ Creating and opening databases
- ⇒ Starting transactions
- ⇒ Creating and retrieving entry points to databases
- ⇒ Creating persistent objects and using them just as you would any objects in a Java application
- ⇒ Creating collections of objects
- ⇒ Querying collections
- ⇒ Optimizing queries by creating indexes on collections
- ⇒ Committing transactions to save changes and possibly retain objects for the next transaction

The best way to develop an application that leverages CFA is to:

- ⇒ Obtain ObjectStore training.
- ⇒ Read ObjectStore white papers about applications that are similar to yours.
- ⇒ Apply the appropriate design principles for your goals.

## Glossary

This glossary defines CFA concepts as well as features supported by ObjectStore.

active persistent object	Applications can read or update only active persistent objects. An active persistent object starts as an exact copy of the object that it represents in the database. The contents of an active object are available to be read and might be available to be modified. If an application updates an active object, the active persistent object is no longer identical to the object in the database that it represents, until the database is updated at transaction commit.
archive logging	Records transaction activity, which provides the information you need to recover modifications made after a backup.
annotations	Byte code instructions required for persistence. The class file postprocessor annotates compiled classes to make them persistence capable.
backup and restore	Copies data to secondary storage, and restores data from secondary storage to primary storage.
CFA server	Also known as ObjectStore server. CFA servers manage physical data on disk, arbitrate among CFA client processes that are requesting objects, and ensure that all clients have consistent views of data.
CFA client	CFA clients provide the interface between your application and the CFA server. Clients apply your business rules to a logical view of your data. For example, in a real time data processing application, a CFA client can perform a query over a body of data that includes new data coming in as well as historical data.
class file postprocessor	See postprocessor.
clustering	Lets you specify where to store an object. For example, if there is a group of objects that you frequently use together you can store them near each other to increase locality. Conversely, you can store objects in different locations to ensure maximum concurrency.
collection	Object that groups together other objects.
compaction	Frees storage space so it can be used by other objects.
database root	Provides a way to associate a name with an object in a database. Applications use database roots to locate one or more persistent objects for performing queries or navigating to other persistent objects. When you make an object the value of a persistent database root, doing so establishes the object as persistent and makes the objects it refers to available for transitive persistence.
deadlock	Two clients are waiting for resources. Each client is waiting for the resources being used by the other client.
deadlock detection	Detects and breaks deadlocks by aborting one of the transactions involved in the deadlock. This causes the victim's locks to be released so that other processes can proceed.

hollow persistent object	Has the same structure and contains the same fields as the object in the database that it represents. However, the fields of the hollow object usually do not contain values corresponding to those values stored in the database. The values for the fields in a hollow object might be null or they might be the values from a previous transaction. A hollow persistent object is a placeholder for an active persistent object. References to hollow persistent objects are valid. A hollow object becomes active, (its fields have valid contents and it has the appropriate database read or write lock) if you read or write one of its fields in a transaction.
JMTL	The Java Middle-Tier Library provides transparent, high-performance storage for Java objects. JMTL caches objects accessed by client-initiated transactions, maintains the consistency and recoverability of the transactional caches, maintains each transaction's required isolation level, and schedules transactions to optimize throughput.
locality	Degree to which objects that are used together are stored near each other on the CFA server.
MVCC	Multi-Version Concurrency Control allows objects that are being accessed by multiple read-only transactions to be simultaneously updated by one update transaction.
notification	Lets a CFA client notify other clients that an event has taken place. Typically, the event is a change in the objects to which other clients have access. Using the notification service, clients can send and receive notifications.
persistence aware	Methods of a class can operate on fields of persistent objects, but instances of the class itself are not persistence capable. Persistence aware classes require special annotations that the class file postprocessor can insert.
persistence capable	Capacity of an object to be stored in a database. If you can store the instances of a class in a database, the class is a persistence-capable class and the instances are persistence-capable objects.
persistent object	Representation of an object that is stored in a database. After an application retrieves an object from the database, the application works with the persistent object in the Java environment. A persistent object always exists in one of three states: hollow, active, or stale. The methods you call on persistent objects can change the state.
postprocessor	Annotates compiled classes so that an application can store instances of those classes in databases. The postprocessor makes classes persistence capable.
queries	Queries return a collection that contains those elements of a given collection that satisfy a specified condition. Queries can be executed with an optimized search strategy, formulated by the query optimizer.
query indexes	The query optimizer maintains indexes into collections based on user-specified keys, that is, data members, or data members of data members, and so on. With these indexes, implemented as B-trees or hash tables, you can minimize the number of objects examined in response to a query. Formulation of

	optimization strategies is performed automatically by the system. Index maintenance can also be automatic.
stale persistent object	Is no longer valid. Its fields have default values or values left over from previous transactions. References to stale persistent objects are not valid. A persistent object can become stale when the transaction in which it was accessed ends.
transactions	Execution of a sequence of statements that operate on objects as a logical unit. That is, the operations performed by the statements are performed all together or not at all.
transitive persistence	Transient objects that are referenced by persistent objects become persistent when the transaction commits. For this to work, the transient objects must be persistence capable.

## ***About Progress Real Time Division***

The Progress Real Time Division provides event stream processing, data management, data access and synchronization products to enable the real-time enterprise. Our products manage and analyze real-time event stream data for applications such as algorithmic trading and RFID; accelerate the performance of existing databases through sophisticated caching; manage and process complex data in the industry's leading object database; and support occasionally connected mobile users requiring real-time access to enterprise applications. The Progress Real Time Division is an operating unit of Progress Software Corporation (Nasdaq: PRGS), a global software industry leader. Headquartered in Bedford, Mass., they can be reached at [www.progress.com/realtime](http://www.progress.com/realtime) or +1-781-280-4000.

[www.progress.com/realtime](http://www.progress.com/realtime)

### **Worldwide and North American Headquarters**

Progress Real Time Division, 14 Oak Park, Bedford, MA 01730 USA Tel: +1 781 280 4000

### **UK Office and Northern Ireland**

Progress Real Time Division, 210 Bath Road, Slough, Berkshire, SL1 3XE England Tel: +44 1753 216 300

### **Central Europe**

Progress Real Time Division, Konrad-Adenauer-Str. 13, 50996 Köln, Germany Tel: +49 6171 981 127

### **France**

Progress Real Time Division, 3 Place de Saverne, Les Renardières B, 92901 Paris la Défense Tel: +33 1 41 16 16 56

© 2006 Progress Software Corporation. All rights reserved. Progress, ObjectStore, Apama and Cache-Forward are trademarks or registered trademarks of Progress Software Corporation, or any of its affiliates or subsidiaries, in the U.S. and other countries. *Java and all Java-based marks are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.* Any other trademarks or service marks contained herein are the property of their respective owners. Specifications subject to change without notice. Visit [www.progress.com/realtime](http://www.progress.com/realtime) for more information.