



Programmer's Reference Guide

EsiObjects V4.2

(c) Copyright 1994 - 2004, ESI Technology Corp, Bolton MA

This document contains the intellectual property of its copyright holder(s) and is made available under a license. If you are not familiar with the terms of the license, please refer to the license.txt file that is a part of the distribution kit.

Information in this document is subject to change without notice. Companies, names and data used in examples herein are fictitious unless otherwise noted. No part of this document may be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose, without the express written permission of ESI Technology Corporation.

Trademarks

EsiObjects is a registered trademark of ESI Technology Corporation.

GT.M is a registered trademark of Sanchez Inc.

DSM, Cache, MSM are registered trademarks of InterSystems Corporation.

Microsoft, Visual Basic, Windows and Windows NT are registered trademarks of Microsoft Corporation.

Table of Contents

PROGRAMMER'S REFERENCE GUIDE	1
ESIOBJECTS V4.1	1
TABLE OF CONTENTS	3
INTRODUCTION	I
DOCUMENT CONVENTION	I
PART 1: CLIENT SERVER OVERVIEW	1
ESIOBJECTS OVERVIEW	2
CLASS DEVELOPMENT ENVIRONMENT	3
APPLICATION RUNTIME ENVIRONMENTS	5
PART 2: OBJECT LIFECYCLE ISSUES	6
USING OBJECTS	7
WHAT IS AN OBJECT?	7
OBJECT INTERFACE.....	8
<i>Primary Interface</i>	9
<i>Factory Interface</i>	10
<i>VariableFactory Interface</i>	10
BUILDING OBJECTS	11
<i>Creating, Preserving and Destroying Objects</i>	11
<i>Creating an Object from a Class</i>	12
<i>Virtual Objects</i>	13
<i>Prebuilt Objects</i>	18
<i>Issues When Building Objects</i>	19
ACCESSING OBJECTS.....	19
USING METHODS.....	20
<i>Definition of Methods</i>	20
<i>Delegating Responsibility to another Object</i>	20
<i>Using Methods with the DO Command</i>	21
<i>Methods and Evaluating Expressions</i>	22
<i>Using Static Methods</i>	22
USING PROPERTIES	23
<i>Properties and Accessors</i>	23
<i>Accessor Input Specification</i>	23
<i>Generated Events</i>	25
<i>Using Accessors</i>	25
USING EVENTS	34
<i>Definition of Events</i>	34
<i>The Event Cycle</i>	35

<i>How an Object Watches for Events</i>	35
<i>How Events are Triggered</i>	36
<i>How the Event Notification is Terminated</i>	36
USING RELATIONSHIPS	37
CREATING AND DESTROYING OBJECTS	37
<i>Object Life Cycle</i>	37
<i>Object Creation</i>	38
<i>Object Preservation</i>	40
<i>Object Protection</i>	40
<i>Object Destruction</i>	41
USING CLASS LIBRARIES	44
<i>Absolute and Virtual Libraries</i>	46
INTEGRATING OBJECTS	47
ELEMENTS OF INTEGRATION.....	47
<i>Object Contracts</i>	47
<i>Object Responsibilities</i>	47
HOW TO INTEGRATE OBJECTS	47
<i>Grouping Integration</i>	47
<i>Event Based Integration</i>	49
<i>Relationship Integration</i>	54
GUIDELINES FOR USING OBJECTS	56
OBJECT LIFE CYCLE.....	56
<i>Object Creation</i>	56
<i>Object Lifetime</i>	57
<i>Object Destruction</i>	61
ADDING INTERFACES TO AN OBJECT	64
OBJECT NAVIGATION.....	65
<i>Creation</i>	66
<i>Nesting</i>	66
<i>Data Sources</i>	66
<i>Events and Messages</i>	68
<i>Domain Names</i>	69
DEFINING OBJECTS	69
<i>The Types of Classes</i>	69
<i>Inheritance</i>	70
<i>Multiple Inheritance</i>	71
<i>Overriding</i>	73
<i>Message Searching</i>	73
<i>Building the Class Hierarchy</i>	74
<i>Avoiding Multiple Inheritance Conflicts</i>	75
<i>Resolving Multiple Inheritance Conflicts</i>	75
PART 3: REUSEABILITY.....	76
USING COLLECTION CLASSES.....	77
WHAT ARE COLLECTION CLASSES?	77
COLLECTIONS PROTOCOL	77
<i>Collections Hierarchy</i>	77
<i>Collection Class</i>	78

<i>Set Class</i>	80
<i>Bag Class</i>	81
<i>Array Class</i>	83
<i>List Class</i>	83
<i>Dictionary Class</i>	84
<i>Log Class</i>	86
<i>Map Class</i>	87
<i>MultiMap Class</i>	90
CHOOSING A COLLECTION CLASS.....	93
CREATING COLLECTION OBJECTS	94
MANIPULATING COLLECTION OBJECTS.....	95
<i>Collection Life Cycle</i>	95
<i>Accessing all Elements in a Collection</i>	96
<i>Iterators</i>	96
<i>Using Iterators</i>	104
<i>Collection Operations</i>	113
USING IMMUTABLE CLASSES	117
WHAT ARE IMMUTABLE CLASSES?.....	117
IMMUTABLE PROTOCOLS	117
<i>Immutable Hierarchy</i>	117
<i>Immutable Class</i>	117
<i>Date Class</i>	117
<i>Interval Class</i>	117
<i>Mvariable Class</i>	117
<i>NameValuePair Class</i>	118
<i>Time Class</i>	118
<i>TimeRange Class</i>	118
<i>TimeStamp Class</i>	118
USING THE DATAMANAGER CLASS	119
WHAT IS A DATAMANAGER CLASS?	119
CREATING AND DESTROYING A DATAMANAGER OBJECT	119
<i>Creating a DataManager</i>	119
<i>Destroying a DataManager</i>	120
THE DATAMANAGER INTERFACE	120
<i>Class Property</i>	120
<i>ControlsData Property</i>	120
<i>CreateElement Method</i>	121
<i>InsertElement Method</i>	121
<i>RemoveElement Method</i>	122
<i>Cardinality Property</i>	122
<i>SelectMatches Method</i>	123
<i>Keys Property</i>	123
<i>AddKey Method</i>	123
<i>RemoveKey Method</i>	124
USING CRITERIA CLASSES	125
WHAT ARE CRITERIA CLASSES?	125
CRITERIA PROTOCOL	126
<i>Criteria Hierarchy</i>	126

<i>Criteria Class</i>	127
USING MIX-IN CLASSES	138
WHAT ARE MIX-IN CLASSES?.....	138
ADDING INTERFACES USING MIX-IN CLASSES.....	138
<i>Accessing Interfaces</i>	138
<i>Major Interface</i>	139
PART 4: EXTERNAL INTERFACE	143
EXTERNAL CALL INTERFACE	145
INITIALIZATION.....	145
API INPUTS	145
<i>CREATE</i>	145
<i>INVOKE</i>	146
<i>SETPROP</i>	147
<i>GETPROP</i>	147
<i>DESTROY</i>	148

Introduction

This guide is designed to assist the EsiObjects programmer build object oriented application systems. It contains the following:

- An overview of EsiObjects.
- Information on how to use the EsiObjects tools.
- How to call EsiObjects from an external M environment.
- Using objects within the EsiObjects environment.
- Description of all database, collection and bulk transfer reusable classes.
- How to integrate objects into an application.
- How to use objects.

Document Convention

EsiObjects documentation uses the following typographical conventions:

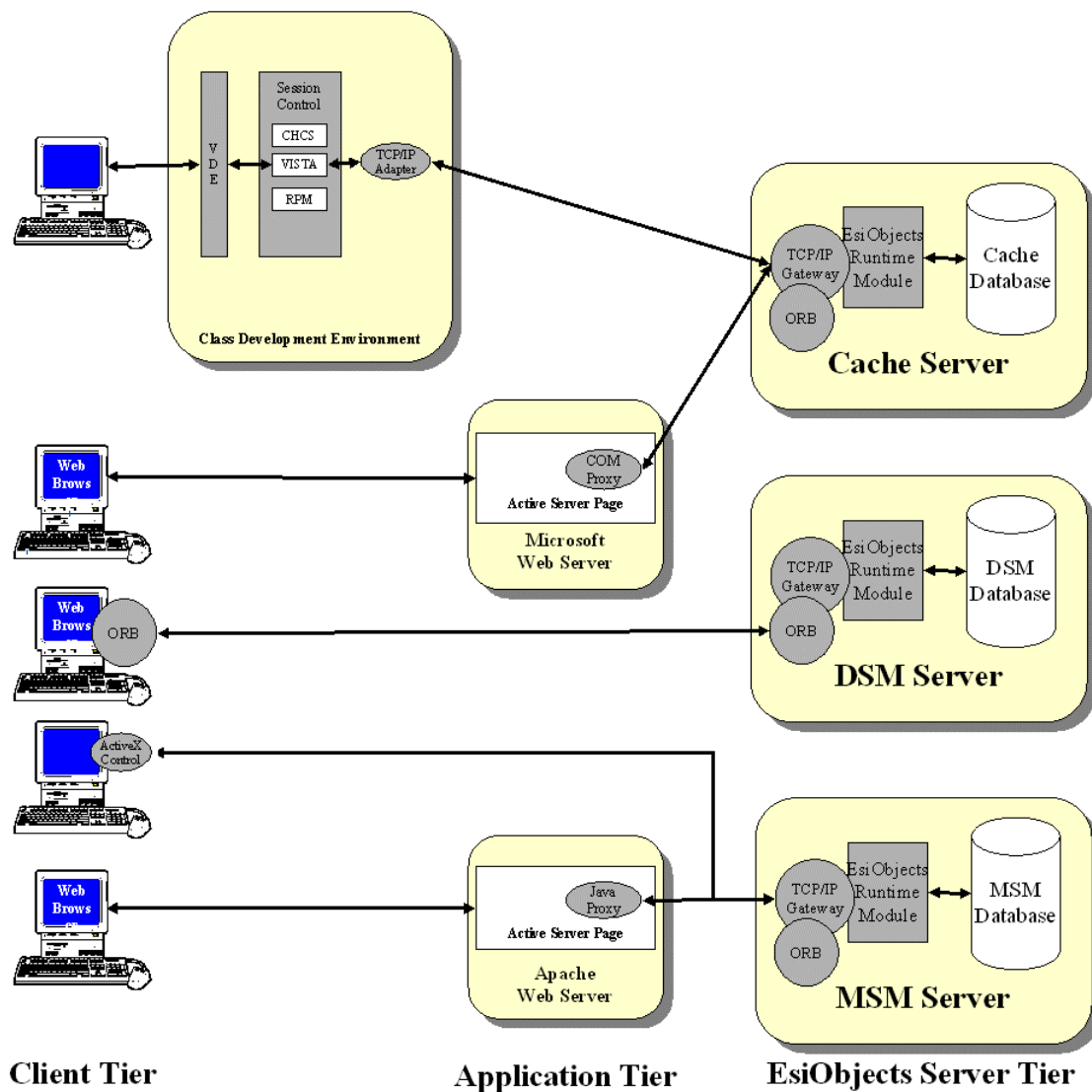
For more information on this subject please refer to the <u>BREAK Command section of this Guide.</u>	Underlined text is used to highlight a reference to another section of this manual or another guide.
<i>Property</i>	In text, italicized words indicate defined terms that are usually used for the first time. Words are also italicized for emphasis.
CREATE	Words in bold and capitalized are EsiObjects commands or keywords.
Set T%Test=I%Pat.Name	This font is used for code examples.

Part 1: Client Server Overview

EsiObjects Overview

EsiObjects (pronounced 'easy objects') is an object oriented database system that runs on the most popular ANSI, FIPS and ISO standard MUMPS (M) systems: DSM, MSM, GT.M and Cache.

Illustrated below are some possible multi-tier EsiObjects configurations. The major components of the EsiObjects system reside on the server tier. They are the **EsiObjects Runtime Module** and **TCP/IP Gateway**.



The **EsiObjects Runtime Module** contains all the components of a state-of-the-art object oriented database system. This module contains support for the **Class Development Environment** and all **Application Runtime Environments**. It implements:

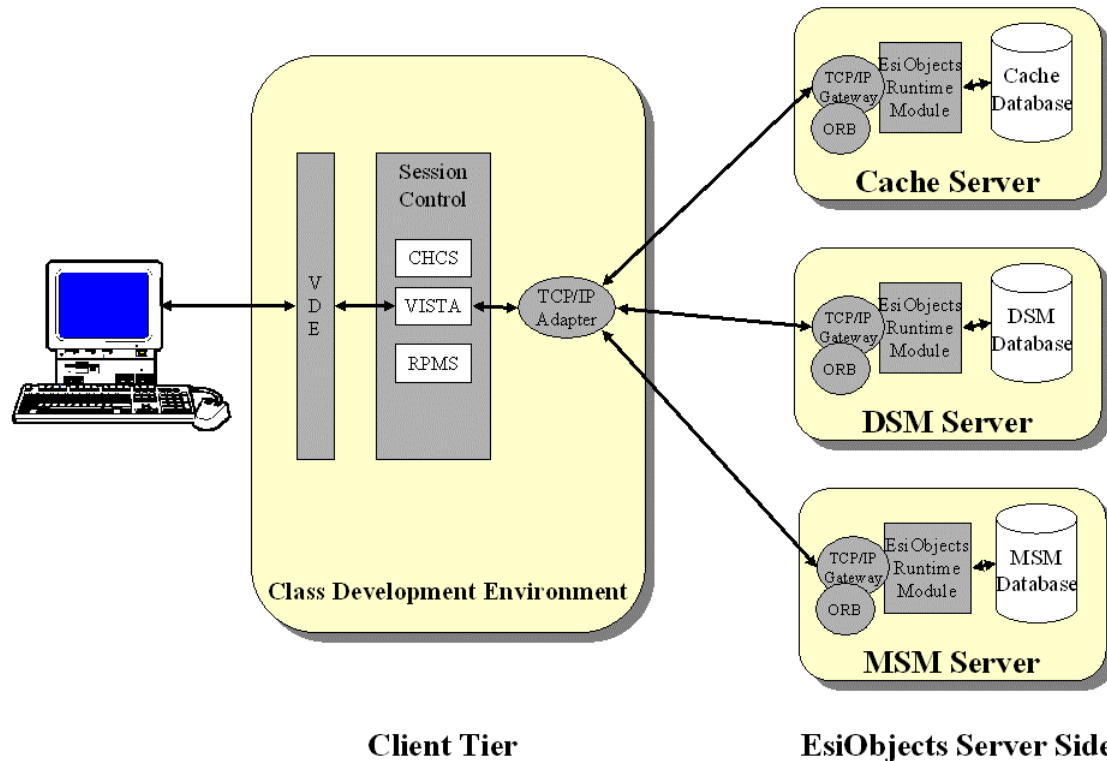
- A robust, comprehensive object model implementation based on the Smalltalk class model. This model implements all the concepts required of a full object oriented system, particularly, encapsulation, inheritance and polymorphic behavior.
- A compiler that implements the 1995 ANSI standard M language and EsiObjects language extensions in support of the object model.
- Numerous linguistic enhancements that evolve the M language into a powerful object oriented language that supports persistent or nonpersistent objects.
- Concrete and virtual libraries are supported to partition classes for convenient access and transfer.
- Full support for the Class model including single and multiple inheritance as well as nested classes.
- Class services partitioned by specific interfaces. Method, property, relationship and event object services that can be used to implement a new application based on real objects or virtual objects that wrap existing M data.
- Full variable scoping that enforces encapsulation, a fundamental requirement of object orientation.
- Bi-directional message service that permits communication between objects. These services include all public class interfaces that allow access to an object's methods, properties and relationships.
- Complete event handling model that permits objects to watch for events that are fired by other objects. The event-handling model is based on a built-in callback mechanism that is available to the programmer as well.
- Numerous predefined classes such as Collections (Lists, Arrays, Set, etc.), Immutables (time and date stamps), Data Manager, etc. that accelerate development through reuse.

The **TCP/IP Gateway** implements an object oriented API into the EsiObjects Runtime Module. It supports all valid message protocols needed to invoke an object's services. It provides the foundation support needed to implement the more sophisticated forms of connectivity supported by EsiObjects. It also provides direct, simple connectivity via ActiveX controls. The COM proxy, Java proxy and CORBA ORB communications components enable applications to run in multi-tier Client Server or Internet based configurations.

Class Development Environment

The diagram below illustrates the **Class Development Environment** that runs on any Windows NT, 2000 or 98 PC. The Class Development Environment is designed to

expedite the development of classes. Variable declaration as well as method, property, event and relationships are developed through editors and wizards. It provides the programmer with all the tools needed to rapidly develop applications.



The **EsiObjects Class Development Environment (CDE)** implements all the necessary browsers and tools needed to implement an object oriented application.

The CDE activity is directed to the server side implementations through a Session Control module. The Session Control module supports connections to multiple server tier M implementations that are running the EsiObjects Runtime module and TCP/IP Gateway. The programmer can have any number of sessions active at once.

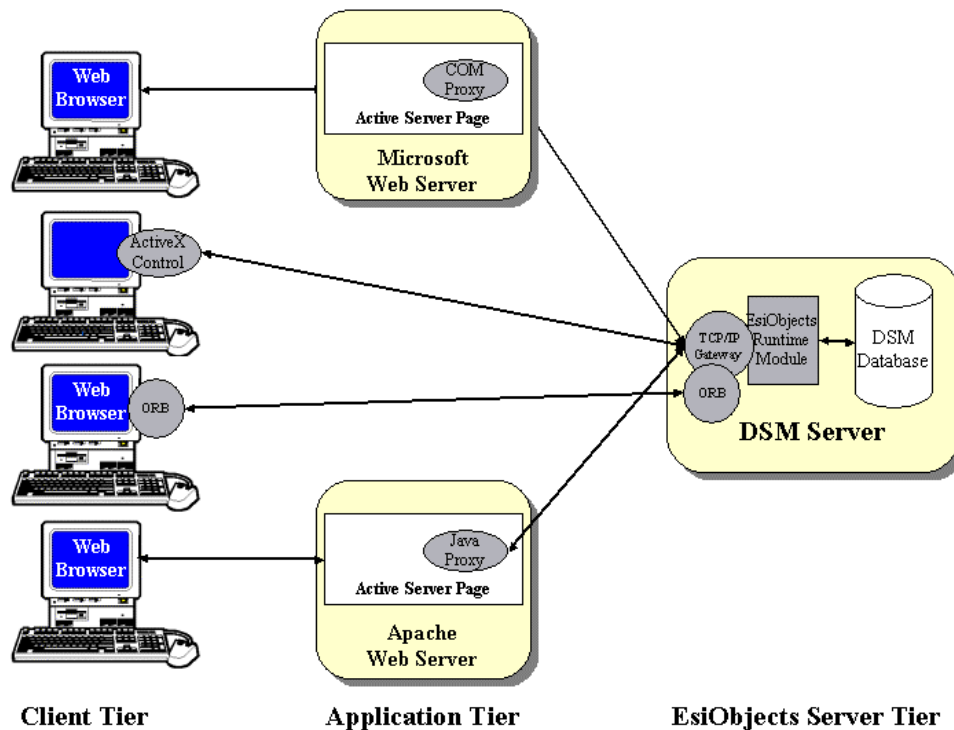
It is important to understand that the client and the server tiers of EsiObjects run independently of each other. They are loosely bound to permit interoperability between different M environments.

Central to the CDE is the Session Browser. It provides access to all libraries and work folders contained in connected sessions with full drag and drop features expected in a modern object oriented development environment. The CDE contains sophisticated Search/Edit functionality as well as an Interactive Debugger and Object Browser. It also provides Export and Import facilities for the transfer of class libraries and all of their components as well as access to traditional Global and Routine facilities.

The On-line Help facility of EsiObjects contains all documents needed to use EsiObjects including an on-line Getting Started Tutorial.

Application Runtime Environments

Illustrated below are the various application runtime connections that are possible with EsiObjects.



On the right is a DSM Server running the EsiObjects Runtime Module, TCP/IP Gateway and a CORBA ORB.

At the top of the diagram is a Microsoft Web Server that contains an Active Server Page (ASP). Within that server page is a COM proxy that permits connectivity to the server side via the TCP/IP Gateway. Active Server Page implementations are accessed via Web Browsers running on a client.

The next diagram down is a simple TCP/IP connection to the server side via an ActiveX control. This control can be used within any popular GUI development environment such as Visual Basic, Visual C++, Delphi, etc.

Next is a client that is running a client ORB that communicates to the ORB running within the EsiObjects environment.

At the bottom of the diagram is the EsiObjects Java Proxy that runs within an server page on an Apache Web Server.

EsiObjects supports multiple middleware implementations. The COM and Java proxies can run in diverse environments.

Part 2: Object Lifecycle Issues

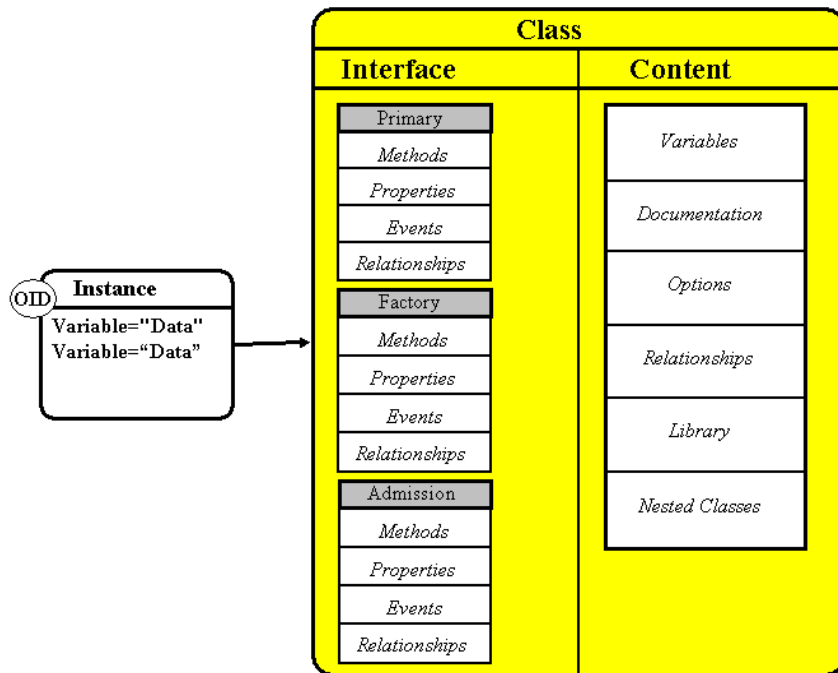
Using Objects

What is an Object?

The diagram below illustrates an object (instance) that was created from the class on its right. An object is a discrete entity that contains data in the form of variables. The combined nature of the data gives the object its **state**. It can be viewed as a symbol table.

Classes are special objects that contain all the definitional information needed to create the instance. When an object is created from a class, it is assigned a unique identifier called an **Object Identifier (OID)**. This gives an object **identity**. Instances always contain an internal pointer to the class that created them. This pointer provides a way for the instance to know what services are available to it. EsiObjects supports the following services: Methods, Properties, Events and Relationships. These services are partitioned into interfaces. Methods are code bodies that give the object its **behavior**. Properties are used to expose the state of the object to the outside. Relationships are used to form links between objects so that they may collaborate with each other. Events may be used to alert other objects of a state change for example.

Hidden from the object user are other things a class is responsible for. The class knows how to create Variables. Each class has a documentation object associated with it. When relationships are defined, they may be one to many and require Collection objects to hold the objects pointed to. This definitional information is stored in the class. Additionally, Nested classes are pointed to if they exist.



When accessing an object's data it must be accessed via a method or property in the object's interface. Because encapsulation is enforced in EsiObjects, it cannot be accessed

directly. Only the methods or properties in the object interface can access the data. Entities that are external to an object (other objects or API's) can communicate with it via messages, thereby indirectly accessing or modifying its data.

The following is a description of the internal characteristics of an object.

- Identity** The identity of an object is a unique identifier that contains the object's location or address. The Object Identifier (OID) represents identity in EsiObjects. The OID of an object is a unique address that defines its exact location. The OID is externally accessible to other objects and can be thought of as a handle to the object. To send a message to an object, you can specify any symbol that contains the OID of an object as the target of a message.
- State** State is the collective value of the object's data. All data in an object and the current values of that data refer to the object's state. If you change the data values, the state of the object changes.
- Behavior** Behavior is the interaction with an object. The behavior of an object is defined by the actions an object can exhibit. From the perspective of an object user, the behavior of an object is exposed through its interface. The behavior of an object can result in any of the following:
- A state change
 - Message generation
 - Event generation

The following section(s) describe the object interface. For more information about messages and events, see the section [Guidelines for Using Objects](#) in the guide.

Object Interface

The object interface is where object's services are found. It contains the methods, properties, relationships and events supported. The interface is the external view into an object. The following are the elements that make up the object interface:

- Methods** Methods perform operations that give the object its behavior. A method is a body of code that performs an operation and may or may not return a value.

Properties A property is an external representation of an internal state of an object. Properties reflect the state of an object and are used to modify the state. Properties provide a safe mechanism so the state of an object can be exposed to the outside world. Properties represent something that a user expects to be a part of an object's physical state, but they are not bound directly to the physical state. The various ways to access the object state (assign, lookup, and so on) can be limited on a property-by-property basis via the implementation of property accessors. Accessors are special methods.

Events An event is something that happens at a given point in time. Objects use events to communicate to interested parties without knowing who they are. All interested parties take out a watch on a particular event. When an object fires that event, the watching objects are informed of its firing through an established callback mechanism that the watcher specified.

There are two major forms of events:

- **Generalized** events indicate that some element of the object has changed (not necessarily its state).
- **Property** events indicate that the state (value) of a property is changing.

Relationships A relationship is a way for objects to know about each other. A relationship has Cardinality, that is, an object can have a relationship established with one and only one other object (one-to-one cardinality) or a one-to-many relationship. Additionally, inverse relationships can be established to make sure the known object knows who know it.

Objects react to requests through their interface and also notify the outside world when a certain criteria in its state have been met.

The previous shows the Primary, Factory and Admissions interfaces for an object. The Primary and Factory interfaces are significant and will be explained in more detail. Within the EsiObjects Class Development Environment, you as a programmer can create any number of interfaces within a class.

Primary Interface

The primary interface may contain a set of methods, properties, relationships and events that are exposed by an object and accessed by a calling object directly or indirectly. The primary interface is the default interface to the object. When a class is first created only the Primary interface is created. Other internal structures and objects are created at that time all well such as Variable and Documentation objects.

Objects communicate with each other via messages. The message syntax contains the path to a specific item (property, method or relationship) within an interface. The structure **Object.[Interface::]Item[(parms...)]** represents the general message syntax of

EsiObjects. **Item** is the property or method name. (**parms...**) is a list of optional parameters. **Interface** is the optional name of the interface the item exists in. **Object** is the OID of an object (associated with a class) that is receiving the message.

Making a reference of **Object.Item** assumes the Primary interface is being used.

Other interfaces can be added to the object to partition its functionality. Often these interfaces are used to hide detail not normally of concern to a general user of the object. Other interfaces also can be used as a mechanism to ensure some common protocol contract.

Factory Interface

The factory interface contains class level methods, properties and events that are exposed explicitly through the message syntax. The factory interface may contain four methods of interest.

CREATE If the CREATE method exists at the time an object is instantiated, it will be executed immediately after the object is created. The name must be spelled correctly and in uppercase. It is useful for initializing the state of an object.

DESTROY If the DESTROY method exists at the time an object is destroyed, it will be executed before the object is removed from the system. The name must be spelled correctly and in uppercase. It is useful for cleaning up around the object before it is destroyed. It can also be used to abort the destruction of the object based on some internal or external condition. See the EsiObjects the [DESTROY](#) command in the [EsiObjects Language Reference Guide](#) for more information.

InitAllSysVars The variable editor creates this method when an instance variable is declared as Initialized (created at object creation time). This method replaces the old InitSysVars. The InitAllSysVars does a full compile of variables up the inheritance tree at compile time. The old method was forced to search at runtime with the Knows message keyword which created performance problems.

InitClassVars The variable editor creates this method when a class variable is declared as Initialized. The programmer may insert extra code to enhance initialization.

VariableFactory Interface

The variable editor inserts this interface when a variable is created as dynamic (created the first time it is referenced). A method is inserted into this interface with the same name as the variable. The method will contain generated initialization code that is executed when the variable is first referenced. Additional initialization code may be added to this method to enhance the initialization of the variable.

Building Objects

The following sections describe the mechanisms available for building objects.

There are two, general types of objects:

- Real
- Virtual

The majority of objects are real. Real objects that contain data and consequently maintain state. However, Virtual objects are much more lightweight. Virtual objects are simply OID's that contain information its services need to perform their operations. They can not store data. Virtual object have classes and services like any real object. They are often used to access structures external to EsiObjects (such as M globals, or text files), or information that can be produced by the system (such as a representation of the time or date).

Creating, Preserving and Destroying Objects

The **CREATE** command is used to create an object. The **DESTROY** command is used to destroy an object.

The timing of object creation can be important depending on the type of object involved and the relationships that the object is to form.

Every object that is created can be destroyed. Generally, when you finish with an object you should invoke the **DESTROY** command for that object. This literally destroys the object if the internal reference count is less than 1, and it can no longer be accessed. The internal reference count can be incremented using the **PRESERVE** command. Any other object that has access to it can preserve the object using this command. This ensures that the object will not disappear before the preserving object has finished with it.

You can create the following types of objects:

Type	Description
Private	Private objects are not designed to be passed outside the process.
Shared	Shared objects are meant to be shared among multiple users.
Child	A child object has the same access (shared or private) and life span attributes as its parent. The default in EsiObjects is to create child objects.
Stack	The object is created within the current call frame. This type overrides all other types. This means that the object does not have to be explicitly destroyed. It will disappear when the current call frame is popped from the stack.

These types of object are created with the **CREATE** command by specifying the specific keywords.

It is valid for private objects to have shared components. However, shared objects cannot have private components. This prevents integrity errors, whereby a shared context might reference a nonexistent private context.

Creating an Object from a Class

You can create objects (instances) of a class with the **CREATE** command. When you create an object you are creating a new instance of the class. The object is created immediately and is initialized to the state you specify with the **CREATE** command (using parameters and initialization properties).

The **CREATE** command lets you accomplish the following:

- You can control the initial state of the object.
- You can ensure that the object is created from a specific class.
- You can control the core attributes of the object (for example, shared or child).

For more information about the **CREATE** command, see the description in the [EsiObjects Language Reference Guide](#).

Requirements for Building an Object from a Class

To build an object from a class, the following requirements must be met:

- The class must exist (you must be able to see the class using the Library Browser).
- You need sufficient resources (disk space and memory) on your system.

Issues When Creating an Object from a Class

The first thing to decide when creating an object is to identify what type of object you want to create. The type of object to create depends on the requirements of your application and depends on the capabilities of the various classes available to you.

Normally an object is created as a child object of the defining object context, which means that an object normally survives until the creating object is destroyed. If the object is going to be shared outside the current context, then use the `Share=1` keyword with the **CREATE** command.

You must determine the initial state that the new object should be in when it is created. This might be as simple as determining the object's defaulted state or as complex as specifying initial contents, relationships, or limits. Specifying the appropriate creation parameters and the initial properties sets the initial state. Note that some properties can be assigned only during object creation.

Examples

The following example shows how to create an instance of a Set that resides in the Base library.

```
CREATE T%Items=Base$Set
```

The following show how to create an Address object, passing it a parameter containing a default state. The parameter will be accepted by the CREATE factory method. It will use it to initialize the I%State instance variable to "MA"

```
CREATE I%CustAddr=Framework$Address("MA")
```

Alternatively, the following example shows how to use the concept of a property to accomplish the same thing. The CREATE accessor would accept the value "MA" as a parameter and bind it to the I%State instance variable.

```
CREATE I%CustAddr=Framework$Address::(State="MA")
```

The following example shows how to create an object that will wrap an external file named test.tmp. The object is persistent.

```
CREATE I%File= Base$AbsSerializationObject:(Share=1):(Name:"test.tmp")
```

Virtual Objects

Virtual objects, frequently used to build an object representation of external information, are objects that have no instance variables. The state of the object is represented by the external structures or entities it wraps. A classic example of this is a "wrapper" object that is typically used to provide an object representation for data in legacy M globals. However, virtual objects might be created for anything external to EsiObjects, such as a text file at the operating system level. The M language's **\$H** special variable is also "wrapped" by the EsiObjects **Base\$TimeStamp** class, which represents the primitive **\$H** format with a rather sophisticated object interface.

Virtual objects encapsulate only a single string's worth of data, stored in the **\$ZVIRDATA** special variable which is available to all services of the class that creates the virtual object. This value must be assigned by the object's **CREATE** method; it cannot be changed later on, once the object has been created. This limits the application of virtual objects to those cases where there is no need to extend the object in the future. They are used to wrap the data as is, providing a façade that has the standard EsiObjects interface but no instance variables (way to store state).

Virtual Object Creation and Destruction

To designate a class as virtual, the class property **Virtual** must be checked on (in the Class's property sheet), and the class must have a unique **Virtual ID**. Uniqueness of virtual ID's is *not* enforced by EsiObjects. It is your responsibility to pick a unique ID in the range **1025** to **65024**.

Virtual ID's used by ESI are in the range between **1** and **1024**. To use the numbers in this range, you must be signed on with the `/Admin` or `/ESI EsiObjects` command line qualifiers.

Virtual objects are created from classes, just like any other object, by using the **CREATE** command. All virtual objects require a **CREATE** method in their factory interface; it is the task of this method to assign the **\$ZVIRDATA** special variable's value. Note that the virtual object cannot have instance variables; its only internal data must be stored in **\$ZVIRDATA**, and this value can never change after the object has been created.

Please note that the **DESTROY** object has no effect on a virtual object, because virtual objects have no symbol table to be removed. (Of course, the virtual object can implement a **DESTROY** method that will destroy its target data.) The only way to remove a virtual object is to **KILL** the variable containing the handle to the virtual object.

However, it may be inappropriate for one object to make assumptions about whether another is real or virtual. For example, a certain class that is virtual today may become an real class in the future. So a reasonable precaution, when eliminating an object, is to both **DESTROY** it and **KILL** the variable containing the handle to the object. If you want to eliminate the handle but have no intention of destroying object's encapsulated data, then simply **KILL** the variable containing the handle to the object. In the case, you must make sure that the object is indexed so it does not get lost in the system.

Finally, note that if a variable containing the handle to an object is scoped within the current method (i.e. **T%** or **P%** variables), then the *variable* will be destroyed automatically when the method terminates. However, this may *not* result in the automatic destruction of any real *object* being referenced by it.

Requirements for Building a Virtual Object

How can you tell whether to use a virtual or an real object? Since its data or reference to data is contained within the **\$ZVIRDATA** string, a virtual object is very lightweight, and virtual objects sometimes enjoy a performance edge over real objects. However, the data string or reference contained by the virtual object is static; it can never change, once the object has been created. This is the primary criterion in determining what kind of object to use.

It is critical to keep in mind that a virtual object does not have access to any dynamic encapsulated data: although it can make use of EsiObjects language features, it cannot use instance variables.

The EsiObjects **TimeStamp** class illustrates one way to use a virtual object. It is used to represent a single **\$H** date/time value. It implements properties such as **Date** and **Time**, which return a formatted string rather than an encoded **\$H** number. It also implements properties such as **Day** and **Month**, which return the day within a month, and month within a year. Each of these properties is calculated dynamically, based on the value of

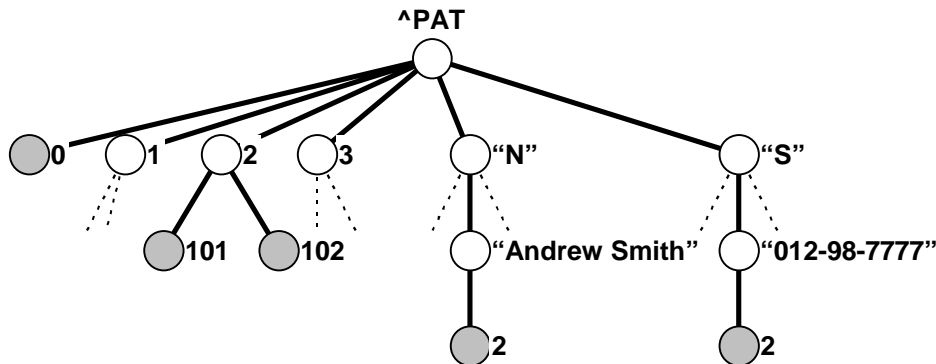
\$ZVIRDATA. All of them implement value accessors; none can be assigned. If you would like to see the implementation details of this somewhat intricate class, you can browse the class **Base\$TimeStamp**.

What if you wanted to create an object to represent an **ASCII** text file in the host operating system? A virtual object might be feasible in this case—**\$ZVIRDATA** could contain the file's name, and standard input/output commands could be used to interact with the actual file. One critical issue would be managing device identifiers for use in input/output operations. A problem would occur, however, if it became necessary to support the operation of renaming the file. In that case, the virtual object would become invalid as soon as the file was renamed. Thus, a virtual object might be appropriate in some cases, but not others.

By definition, virtual objects do not have instance variables. Although you can create instance variables for a class that has been marked virtual, they have no meaning in an executable context.

Virtual Objects Example

Patient File Example—Global Structure



Global "wrapper" objects are the classic examples of virtual objects. Let's imagine an (extremely simplified) legacy M global database, containing patient data. The following global listing shows the global with only three patients.

```

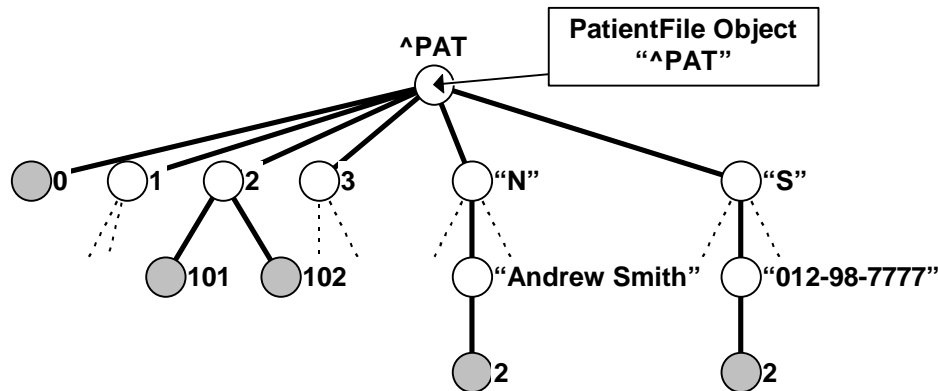
^PAT(0)="3"
^PAT(1,101)="Doe, Jane W^123-45-6789^F"
^PAT(1,102)="22 Beacon St.^Boston^MA^02134"
^PAT(2,101)="Smith, Andrew P^012-98-7777^M"
^PAT(2,102)="33 Water St. #222^Texarkana^TX^54321"
^PAT(3,101)="Aaron, Greta^999-99-9999^F"
^PAT(3,102)="99 Ward St.^Waitsfield^VT^05673"
^PAT("N","Aaron, Greta",3)=" "
^PAT("N","Doe, Jane W",1)=" "
^PAT("N","Smith, Andrew P",2)=" "
^PAT("S","012-98-7777",2)=" "
^PAT("S","123-45-6789",1)=" "
^PAT("S","999-99-9999",3)=" "

```

Each patient in the file is represented by a sequentially numbered array node (**1** through **3**). As new patients are added, they will receive higher numbers, i.e. the next patient will go under **^PAT(4)**. The global implements two cross-references, "**N**" for names and "**S**" for social security numbers.

Each patient entry in the global spans two array nodes, **101** and **102**, each having its own **\$PIECE** layout. The format of node **101** is **Name^SSN^Sex**, and of **102** is **Street^City^State^ZIP**.

Patient File Example—PatientFile Object



The Patient Global **^PAT**, shown in the previous section, is wrapped by two classes of virtual objects. The first class, **PatientFile**, represents the patient file as a whole, and the second, **Patient**, represents a single patient object.

The **\$ZVIRDATA** special variable of the **PatientFile** object contains a global reference to the root of the file—"**^PAT**". This object is responsible for managing the overall file's global structure, and for providing access to virtual **Patient** objects representing data within the file.

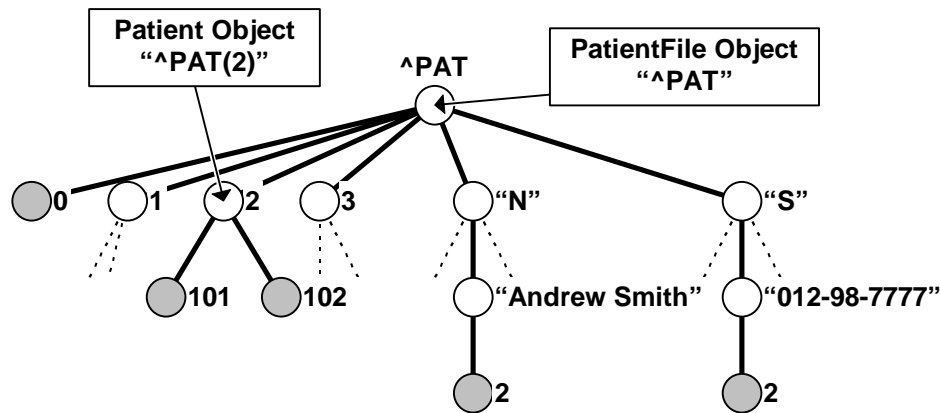
If an external object requests access to an individual patient, the **PatientFile** creates a virtual **Patient** object to represent this data, as follows:

```

;Property - Rapper$PatientFile - Primary::Patients(Value)
; Return matching virtual Patient object, or "" if there is no such
;patient.
Input: (
  P%Ssn ; SSN is input.
)
IF $get(P%Ssn)="" QUIT "" ; SSN must not be null.
SET T%Id=$order(@$ZVIRDATA@("S",P%Ssn,""))
IF T%Id="" QUIT "" ; No such SSN on file.
SET T%PatientGlobal=$name(@$ZVIRDATA@(T%Id))
CREATE T%Patient=Rapper$Patient(T%PatientGlobal)
QUIT T%Patient ; Return patient object.
  
```

Note: If the same request is made again later, the process will be repeated with no problems—remember, virtual objects do not contain data; therefore two identical virtual objects are no different from one virtual object.

Patient File Example—Patient Object



In this example, the **Patient** object represents a single patient's information. Its **\$ZVIRDATA** special variable contains a global pointer to the array node where the patient's data is stored. For example, a patient named Andrew P. Smith's information is stored under **^PAT(2)**:

```
^PAT(2,101)="Smith, Andrew P^012-98-7777^M"
^PAT(2,102)="33 Water St. #222^Texarkana^TX^54321"
```

Normally it is the responsibility of the **PatientFile** object to create virtual **Patient** objects. For testing purposes, however, this object might be created with the following **CREATE** command from the **Xecute Shell**:

```
CREATE I%TestPatient=Rapper$Patient(" ^PAT(2) ")
```

The **CREATE** command shown above invokes the **Patient** class's **CREATE** method, which might look as follows:

```
    ; CREATE method for Patient object.
Input: (P%Global) ; Input is the global location.
    set $ZVIRDATA=P%Global
    quit
```

Notice how simple this is: since virtual objects don't contain any data, it's really not much work to create one.

It is beyond our scope to show all the implementation details of this imaginary **Patient** class. But let's suppose that the virtual **Patient** object implements a separate property for each of the fields in the database's **Patient** record. In that case, the **Value** accessor of the **City** property would look like this:

```
    ;Property - Rapper$Patient - Primary::City(Value)
Input:(
    P%Subs...
)
;
IF P%Subs QUIT "" ; No subscripts allowed.
QUIT $piece($get(@$ZVIRDATA@(102)), "^", 2)
```


The above accessor method is invoked whenever the virtual **Patient** object's **City** property is referenced. The **Assignment** accessor of the **City** property would look like this:

```

;Property - Rapper$Patient - Primary::City(Assign)
Input:(
  P%Value,
  P%Subs...
)
;
IF P%Subs QUIT 0 ; No subscripts allowed.
SET $piece(@$ZVIRDATA@(102),"^",2)=P%Value
QUIT 1 ; Successful completion.

```

The above accessor method is invoked whenever the virtual **Patient** object's **City** property is assigned with the **SET** command. (Setting data into a database using the Assign accessor assumes that the data has been validated and normalized, if required.)

Note: Certain properties of the Patient object, such as **Name**, exist both in the patient record, which is the responsibility of the **Patient** object, and in the global's indexes, which are the responsibility of the **PatientFile** object. The assignment accessor of the **Name** property must therefore involve communication between two virtual objects. Strategies for dealing with such cases are beyond the scope of this discussion. In general, however, the **Patient** object will need to generate a virtual **PatientFile** object for the purposes of such communication.

Prebuilt Objects

Prebuilt objects are objects that you do not have to build. There are two types of prebuilt objects:

Persistent objects Persistent objects are always resident and are always available. EsiObjects supplies some persistent objects. Also, you can create your own persistent object by using the **Share=1** option on the **CREATE** command.

System-defined objects System-defined objects are provided with EsiObjects. The following special variables point to examples of system-provided prebuilt objects:

- **\$ENVIRONMENT**
- **\$LIBRARY**
- **\$SELF**
- **\$SYSPool**

For more information about these special variables, see the [Special Variables](#) section of the [EsiObjects Language Reference Guide](#).

Issues When Building Objects

Generally it is the responsibility of the creator of an object to ensure that it is deleted when it is no longer in use. The exception is objects that are creating objects on behalf of other objects (for example, factories). Under some circumstances, a factory can retain ownership of the objects it creates.

You can reference objects only while they are still alive. Object creation is not a guaranteed operation and can result in a run-time error. An error occurs most often as a result of resource depletion or mismatched parameters. Checking on the existence of an object can be accomplished using the **\$EXIST** or **\$INFO** functions. For more information about these functions, see the [Functions](#) section of the [EsiObjects Language Reference Guide](#).

By default all objects are created with a sharing context equal to the context of the object's creator. For applications, which generally last for the life of the process, an object is created with the keyword `Share=0` (local to the process and not persistent) option with the **CREATE** command by default. If the object is to be shared (by either saving a pointer to it in a global or by using the `Share=1` option), then it should be created with the `Share=1` option. There is no mechanism to alter the `Share` state of an object. Persistent objects are stored in M globals and non-persistent objects are stored in local arrays.

When you no longer need an object, use the **DESTROY** command to destroy the object.

Accessing Objects

Sending the object a message accesses an object's services. A message typically requests an object to do something. The object performs the operation if it exists within its interface. If the service returns a value, it will be sent back to the object.

The following are ways to access an object:

- By **action** - This form of access is a request to perform some action. It is always an invocation of a method, usually by using the **DO** command, for example:

```
DO T%Obj.InsertElement(T%Element)
```

- By **value** - This form of access is a request for some information from an object. The request is either a property or a method that calculates a value and is invoked in an expression, for example:

```
SET T%V=T%Obj.Name
```

- By **chained access** - This form of access is executed from left to right. The following are examples:

```
DO T%List.FindPatient("Doe, John").Sex
SET T%V=T%List.FindPatient("Doe, John").Sex
```

In the example above, the `T%List` temporary variable contains the OID of a list object that contains a list of patients. The `FindPatient` method is a service on that object that looks up the patient specified as the parameter and returns the OID of the patient. That OID is then used to access the `Name` property on the patient object returning the patients `Sex`.

When objects are accessed, parameters may or may not be used. This depends on the request.

The following example shows how to access an object by **action**. First a customer list object is created using the List collection in the Base library. Then an element is added to the List object using the InsertElement method.

```
CREATE I%CustList=Base$List
DO I%CustList.InsertElement("ACME Tool Company")
```

The following example shows how to access an object by **value**. Using the object created in the last example, the Cardinality value is returned.

```
SET T%Card=I%CustList.Cardinality
```

Using Methods

Definition of Methods

A method is a code body that performs some operation for the object and may or may not return a value to its caller via the **\$RETURN** special variable. Methods are created and reside within an interface.

Delegating Responsibility to another Object

Often in object oriented programming it is necessary to create façade objects that hide complex operations. These façade objects typically collaborate with other objects that share in the façade objects responsibility to the calling object. Although the façade object is totally responsible to the calling object, it may very well *delegate* that responsibility to another object. The problem at this point becomes, how do you transfer all the information passed into the object to the helper object. EsiObjects supports the following two ways to do this:

- The **GOTO** Command
- The **\$DELEGATE** function

Delegating Responsibility using the GOTO Command

The **GOTO** command is used to invoke a method by **unilateral delegation**, transferring control to a method of an object. The **GOTO** command can be useful when the current method wants to *delegate* to another object, or to another method of the same object. (Many programmers prefer to avoid **GOTO** to a label within the body of a method or routine, but delegation is a different case.)

The following is the syntax for invoking a method with the **GOTO** command as a form of delegation:

```
GOTO Object.Method
```

```
GOTO Object.Method(Parameters)
```

The following example shows how to delegate a request to the parent object.

```
GOTO I%Parent.Request(P%Param1)
```

Using the **GOTO** command to unilaterally pass control to an object means that control will not be returned to the object like it would be if you used the **DO** command. Additionally, using the **GOTO** command for delegation forces you to select and pass the proper parameters. There is additional overhead in this approach. However, if delegation to another object simply means passing the entire context of the current object to another object, you should always use the **\$DELEGATE** function. This approach is explained in the next section.

For more information about the [GOTO](#) command, see the [EsiObjects Language Reference Guide](#).

Delegating Responsibility using the \$DELEGATE Function

The **\$DELEGATE** function has been specifically implemented in EsiObjects to make the process of delegating an object's call context to another object simple and fast.

Assume that you have a method that is merely a façade and its responsibility is to pass the caller context to a helper object that is actually responsible for processing the request. The simplest approach is to use the following construct:

```
Quit $Delegate(T%HelperObject)
```

The temporary variable `T%HelperObject` contains the OID of the helper. This construct simply passes the entire calling context (via an internal pointer switch) to the current context making it available to a method of the same name in the helper object.

Sometimes it is necessary for the façade object to do some processing based on what the helper object has accomplished. In this case the responsibilities are shared. The following construct can be used in this scenario:

```
Set T%Results=$Delegate(T%HelperObject)
```

In this case, `T%Results` variable would contain any return value for further processing.

Using Methods with the DO Command

Usage

Invoking a method with the **DO** command is useful when you want an object to perform some operation but you do not care about any possible return values.

The following is the syntax of the **DO** command when invoking a method:

```
DO Object.Method
```

```
DO Object.Method(parameters)
```

For more information about the [DO](#) command, see the [EsiObjects Language Reference Guide](#).

Examples

The following example asks an object to update its current state.

```
DO T%Object.UpdateData
```

The following example shows how to have an object move across the screen.

```
FOR T%X=1:1:200 DO T%Graphic.MoveTo(T%X)
```

Methods and Evaluating Expressions

Usage

Methods also can be invoked in the context of an expression. These types of methods return a value. The value returned is generally one of the following:

- Success code
- Result

The following is the syntax for invoking a method as an expression:

```
SET T%Vat=T%Object.Method
```

```
SET T%Vat=T%Object.Method(Parameters)
```

Methods invoked as an expression return a value. This value is related directly to the requested operation. For some methods, the value is a success code that indicates if the operation could be done. For other methods, the return value is the result of the operation, which can vary from simple scalar data to a reference to an object.

For more information about evaluating expressions, see the [Using Expressions](#) section of the [EsiObjects Language Reference Guide](#).

Examples

The following example opens a file with the requested file name.

```
IF 'T%File.Open("test.tmp","R")  
DO $ENVIRONMENT.Assert("Cannot open test.temp")
```

The following example finds the number of elements in a collection.

```
SET T%Total=I%MySet.Cardinality
```

Using Static Methods

Static methods are methods that can be executed without making reference through an instance of a class. The method can be executed by referencing the class directly. Static methods are useful where entry point objects bound to a domain variable (O%) are typically required. To access a static method, you must declare it as static. That can be done by bringing up the property sheet on the method and clicking the Static checkbox on the General tab sheet.

Examples

Normally an entry point object would be bound to a domain variable (O%). For example, assume you had a Database object bound to O%Database and you wanted to run the Initialize method in the Initialization interface. You would have to do it as follows:

```
Create O%Database=MyLibrary$Database
Do O%Database.Initialization::Initialize
```

This approach is valid, however, it requires instantiating a Database object and binding it to the O%Database variable. Additionally, if the object is a singleton, enforcing that becomes problematic.

To avoid this, simply declare the Initialize method as Static. Now you can access it directly as follows:

```
Do MyLibrary$Database.Initialization::Initialize
```

Using Properties

Properties and Accessors

A property expresses the outside view of the state of an object. Properties in EsiObjects are subdivided into discrete code bodies called accessors. An accessor is a special purpose method and controls access to the state of the object. An accessor is strongly bound to the EsiObjects language. For example, the Assign accessor is used when a property is assigned a value via the **SET** command. The Value accessor is used when property is used to produce a value. The following sections describe the different types of property accessors supported.

Accessor Input Specification

Listed below is a table of Input specifications for each property accessor.

Accessor	Input Specification
Assign	Input:(Value,[P ₂ -P _n ,P _{n+1} ...])
Create	Input:(Value,[P ₂ -P _n ,P _{n+1} ...])
Value	Input:([P ₁ -P _n ,P _{n+1} ...])
Kill	Input:([P ₁ -P _n ,P _{n+1} ...])
\$Order	Input:(Direction, [P ₂ -P _n ,P _{n+1} ...])
\$Get	Input:(Default, [P ₂ -P _n ,P _{n+1} ...])
\$Data	Input:([P ₁ -P _n ,P _{n+1} ...])
\$Query	Input:(Direction, [P ₂ -P _n ,P _{n+1} ...])
\$Normalize	Input:(Value, [P ₂ -P _n ,P _{n+1} ...])

\$Valid Input:(Value, [P₂-P_n,P_{n+1}...])

Each accessor has a specific structure that is explained in the [Using Accessors](#) section of this guide. The parameters in the Input Specification must adhere to the following rules:

Rule 1

The first position in the Input Specification is reserved for those accessors that must pass in a value in order to function appropriately. This parameter is called a System parameter. For example, within your code you may have the following line:

```
Set T%Object.Name=T%Name
```

Where T%Object holds a person object's OID and T%Name holds a persons name. In this case, the compiler will generate code that will pass the content of T%Name into the Assign accessor of the Name property. Consequently, the first position of the Input Specification would look like the following.

```
Input:(
    P%Value            ;Value from right side of = in Set command.
)
```

Rule 2

If the ... parameter syntax is used, it must be the last parameter in the list. The ... syntax permits the caller to pass in a list of parameters. If this syntax is used in the Input Specification as the last parameter, this means that all parameters passed in from this position on will be put into a list that has the name specified. For example, assume you have the handle to an object that is a calculator and you want it to add two or more numbers. The call to the calculator object would look like the following:

```
Set T%Total=T%Object.Add(1,30,900,22,4,67)
```

The code body of the Add method would look like this:

```
Input:(
    P%List...
)
Set T%Sum=0
For T%Idx=1:1:P%List Set T%Sum=T%Sum+P%List(T%Idx)
Quit T%Sum
```

Rule 3

All other parameters must go between these two special cases if they exist.

More Examples

The following example will invoke the Value accessor of the Name property to insert the value (Doe, John) into the Tax and Archive indices.

```
SET I%Index.Name("Tax", "Archive")="Doe,John"
```

The Name properties Input specification would look like:

```
Input: (P%Value,P%Indices...)
```

where P% Value will hold "Doe,John" and an array P%Indices will be created that looks like the following:

```
P%Indices=2
P%Indices(1)="Tax"
P%Indices(2)="Archive"
```

Generated Events

Within the EsiObjects system, an object can generate events and other objects can watch for those events. There are two accessors that generate events: Assign and Kill. Anytime a property value is assigned or killed explicitly (or implicitly through the Destroy object command), the appropriate event is triggered. See the **Assign** or **Kill** accessor for specific information on generated events.

Using Accessors

Value Accessor

The **Value** accessor is used to find the value of a property. It is the most commonly used accessor. Most properties implement the Value accessor. By convention the value obtained from this accessor is also valid for the Assign accessor.

Message Syntax

The following is the syntax for a Value accessor:

```
SET T%Val=Object.Property
SET T%Val=Object.Property(parameters)
```

In the following example, an object is created from the Customer class and its handle is stored in the instance variable I%Cust. Next the Value accessor of the Title property of the Customer object is access and returns the title of the customer, binding it to the T%CustTitle temporary variable.

```
CREATE I%Cust=Framework$Customer
SET T%CustTitle=I%Cust.Title
```

The parameters of the **Value** accessor are optional. It does not have a System parameter in the first position, therefore Rules 2 and 3 are applicable.

Input Syntax

```
Input:([P1-Pn,Pn+1...])
```

Typically the Value accessor does not have a parameter list. The simplest Value accessor code could look like the following:

```
;
Q I%Title ;Simply return the value of I%Title
```


Assign Accessor

The **Assign** accessor is used to assign a value to a property. Many objects allow their properties to be assigned. The assign accessor is one means by which a variable value can be added or modified within an object.

The Assign accessor may also be used in lieu of the Create accessor when passing property values in on the Create command. See the [CREATE](#) command in the [EsiObjects Language Reference Guide](#) for more detailed information.

Message Syntax

The following is the syntax for an Assign accessor:

SET Object.Property=Value

SET Object.Property(Parameters)=Value

The following example creates a customer object and binds it to the I%Cust instance variable. Then the Assign accessor is invoked to set the property to the string on the right side of the equals sign.

```
CREATE I%Cust=Framework$Customer
SET I%Cust.Title="ABC Widget Company"
```

The Assign accessor code could look like the following:

```
Input:(P%Title) ;Assign value comes in bound to P%Title
;Note that pass back false (setting $RETURN=0) on an Assign
;accessor forces an error.
;If a value not passed in, then force an error and return
QUIT: '$DATA(P%Title) 0
SET I%Title=P%Title ;Set the customer title to the value
QUIT 1 ;Quit indicating success
```

The **Assign** accessor requires that Rule 1 to be adhered to. The first parameter must be a System parameter and must identify a variable to hold the value to be assigned. If the second and subsequent parameters are specified, they must adhere to Rules 2 and 3.

Input Syntax

Input:(Value,[P₂-P_n,P_{n+1}...])

Where:

Value is the value assigned to the property.

Generated Events

Note: Setting the \$RETURN special variable to 0 will force an exception to occur, resulting in an error message.

By default, the \$Return value is set to 1 (true) before the accessor is executed. During the execution of the accessor, a PRESET event is generated to alert any objects watching this or all properties that the set is about to begin. Once the properties value has been assigned, the SET property event is generated.

If for any reason it is determined that the set should not happen, simply setting \$Return to 0 (false) or QUIT 0 will cause a SETREJECT event to be generated.

Create Accessor

Some properties of an object can be assigned during object creation. Generally, the **Create** accessor is used for this purpose. If the **Create** accessor is not defined, then the **Assign** accessor will be used.

Message Syntax

The following is the **CREATE** command syntax for using the Create accessor:

CREATE Var=Library\$Class::(Property=Value)

CREATE Var=Library\$Class::(Property(Parameters)=Value)

The following example creates a Customer object bound to the T%Cust temporary variable with the CreditRating property set to 1. The Create accessor will be accessed.

```
CREATE T%Cust=Framework$Customer:: (CreditRating=1)
```

The Create accessor code could look like the following:

```
Input: (I%CreditRating=1)
      ;Default instance variable to 1 if not defined
      ;If defined on input, parameter passing will set it automatically.
QUIT
```

Like the **Assign** accessor, the **Create** accessor requires the first parameter to be a System parameter. The first parameter identifies a variable holding the value. Rules 1, 2 and 3 are applicable.

Input Syntax

Input:(Value,[P₂-P_n,P_{n+1}...])

Where:

Value is the value assigned to the property.

Kill Accessor

The **Kill** accessor is used when a property appears as a part of a **KILL** command. Applying the Kill command on an object's property results in the execution of the property Kill accessor. The accessor will determine what to do with the particular property. The Kill accessor gives the programmer control over the destiny of the property value. For example, if the property is used to manage an instance variable containing an OID of an external object the Kill accessor can simply destroy the object or store it away in a trash container.

Message Syntax

The following is the syntax for a Kill accessor:

KILL Object.Property

KILL Object.Property(Parameters)

The following simple example illustrates how the MailStop property of an Address instance would be deleted.

```
CREATE T%CustAddr=Framework$Address
KILL T%CustAddr.State
```

The Kill code above would invoke the Kill accessor of the State property. The code below represents what the accessor executes. In this case it simply kills the instance variable since it contains a pointer to a state object. The accessor could be embellished to actually store the pointer away in an audit object before actually killing it. The Kill accessor permits numerous scenarios.

```

;
KILL I%State

```

The **Kill** accessor allows optional parameters and does not require a System parameter. Rules 2 and 3 are applicable. Typically the KILL accessor does not have a parameter list. However, it could have a parameter that specified a specific trash can object to put the an object in.

Input Syntax

Input:([P₁-P_n,P_{n+1}...])

Generated Events

By default, the \$Return value is set to 1 (true) before the accessor is executed. During the execution of the accessor, a PREKILL event is generated to alert any objects watching this or all properties that the kill is about to be executed. Once the property has been killed, the KILL property event is generated.

If for any reason it is determined that the kill should not happen, simply setting \$Return to 0 (false) will cause a KILLREJECT event to be generated. Additionally, a DEAD event is generated when an object is destroyed.

\$Get Accessor

The \$Get accessor is used to find the value of a property and provide a default if so desired. An optional default value can be provided when the property does not have a value. The \$Get accessor is invoked when the property is used in the \$GET function. Often, the \$Get accessor normalizes the default value through the default specification.

Message Syntax

The following is the syntax for a \$Get accessor:

SET Var=\$Get(Object.Property,Default)

SET Var=\$Get(Object.Property(parameters),Default)

In the following example, an Address object is created and bound to the T%CustAddr temporary variable. Then, the State property is accessed via the \$GET accessor. The default State abbreviation "MA" is passed in.

```

CREATE T%CustAddr=Framework$Address
SET T%State=$Get(T%CustAddr.State, "MA")

```

A possible \$Get accessor is implemented below. The default value "MA" is passed into the accessor code body via the T%Default temporary variable. If it was not specified, it will default to "". If the instance variable does not exist or it is not an object, the default is returned. If not, then the value is returned via a Key property (States are stored as shared objects).

```

;Note: the input default value is defaulted is not specified.
Input:(T%Default="MA")
;Return default if I%State doesn't exist or not an object.
QUIT: '$EXIST($GET(I%State)) T%Default
;Return the existing value.
QUIT I%State.Key

```

The **\$Get** accessor requires a System parameter in the first position with optional parameters following it. The first parameter identifies a variable holding the default value for the \$Get to be applied to the property. Rules 1, 2 and 3 are applicable.

Input Syntax

Input:(Default, [P₂-P_n,P_{n+1}...])

Where:

Default value is the second argument of the \$GET.

\$Order Accessor

The **\$Order** accessor is used when a property is the argument to the **\$ORDER** function. This often occurs when a property exposes some collection.

Message Syntax

The following is the syntax for a \$Order accessor:

SET Var=\$Order(Object.Property(parameters),Direction)

Assume that a customer index object pointed to by the I%CustList instance variable contains an array of all customers a company has. The array is stored in an instance variable I%Index where the first subscript is the customer name and the value of the node is the OID of the customer object.

The code below creates a CustIndex object and binds it to the temporary variable T%CustList. It then performs a Next operation on the customer index object returning the OID of the next customer beyond the last operation.

```

CREATE T%CustList=Customer$CustIndex
SET T%NxtCus=$Order(I%CustList.Next)

```

The following code implements the \$Order accessor.

```

Input:(T%Direction=1) ;Default direction forward.
;Do a $Order on the I%Index array in the direction specified.
SET T%Nxt=$O(I%Index(I%LastCust),T%Direction)
;Set the last customer to the one just found
SET I%LastCust=T%Nxt
;If end of list return null, else return the IOD
QUIT $Select(T%Nxt="":",1:I%Index(T%Nxt))

```

The **\$ORDER** function requires that the first parameter be a System parameter with optional parameters to follow. The first parameter is the direction (1 for forward and -1 for backwards) of the order. Rules 1, 2 and 3 are applicable.

Input Syntax

Input:(Direction, [P₂-P_n,P_{n+1}...])

Where:

Direction is the second parameter of the \$ORDER, either 1 or -1.

\$Query Accessor

The **\$Query** accessor is used when a property reference appears in a **\$QUERY** function. The **\$QUERY** function traverses the leaf nodes of a tree that have values.

Message Syntax

The following is the syntax for a \$Query accessor:

SET Var=\$Query(Object.Property(parameter))

Assume that a customer index object pointed to by the I%CustList instance variable contains an array of all customers a company has. The array is stored in an instance variable I%Index where the first subscript is the customer name and the value of the node is the OID of the customer object.

The code below creates a CustIndex object and binds it to the temporary variable T%CustList. It then performs a NextValue operation on the customer index object returning the OID of the next customer beyond the last operation.

```
CREATE T%CustList=Customer$CustIndex
SET T%NxtCus=$Query(I%CustList.NextValue)
```

The following code implements the \$Query accessor.

```
Input:()
;Do a $Query on the I%Index array in the direction specified.
SET T%Nxt=$Query(I%Index(I%LastCust))
;Set the last customer to the one just found
SET I%LastCust=T%Nxt
;If end of list return null, else return the IOD
QUIT $Select(T%Nxt="":",1:I%Index(T%Nxt))
```

Input Syntax

Input:([P₁-P_n,P_{n+1}...])

Where all parameters are optional.

\$Data Accessor

The **\$Data** accessor is used to determine if a property exists.

Message Syntax

The following is the syntax for a \$Data accessor:

SET Var=\$Data(Object.Property(Parameters))

In the following example, an Address object is created and bound to the T%CustAddr temporary variable. Then, the State property is accessed via the **\$GET** accessor. The default State abbreviation "MA" is passed in.

```
CREATE T%CustAddr=Framework$Address
SET T%State=$DATA(T%CustAddr.State)
```

A possible \$Data accessor is implemented below that simply returns the \$Data value of the I%State instance variable.

```
;Return $Data value of I%State.
QUIT $DATA(I%State)
```

The **\$Data** accessor does not require that the first parameter be a System parameter. It will take optional parameters. Rules 2 and 3 are applicable.

Input Syntax

Input:([P₁-P_n,P_{n+1}...])

\$Normalize Accessor

The **\$Normalize** accessor is invoked when a property is used in a **\$NORMALIZE** function. This accessor is used to transform the property value from an external into an internal value for storage. For example, the numbers 0 and 1 are often used internally to store the external values of No and Yes or Off and On respectively. The \$Normalize function lets the programmer engage the \$Normalize accessor.

Message Syntax

The following is the syntax for a \$Normalize accessor:

SET Var=\$Normalize(Object.Property,Value)

SET Var=\$Normalize(Object.Property.Value(Parameters),Value)

The following example illustrates how the \$Normalize function would normalize a Social Security Number to an internal form that does not have the hyphens embedded. First an instance of Employee is created and bound to the T%Employee temporary variable. Next, the SSN property of the Employee object is accessed within the context of the \$Normalize function. The \$Normalize function returns the normalized value of the SSN.

```
CREATE T%Employee=Framework$Employee
S T%SSN=$NORMALIZE(T%Employee.SSN, "555-55-5555")
```

The following code implements the \$Normalize accessor.

```
Input:(T%SSN) ;Value passed in from $Normalize function.
;The SSN is assumed to be validated to the form 3N1"-2N1"-4N
;Return a hyphenless SSN
QUIT $TR(T%SSN, "-", "")
```

The \$Normalize accessor requires that the first parameter be a System parameter. Optional parameters may follow. The first parameter is the value to be normalized. Rules 1, 2 and 3 are applicable.

Input Syntax

Input:(Value, [P₂-P_n,P_{n+1}...])

Where Value is the value to Normalized.

It returns the normalized form of the value.

\$Valid Accessor

The \$Valid accessor is used when a property is used in a \$VALID function. It is used to determine whether a value is correct before assigning the value to a property. The \$Valid function lets the programmer engage the \$Valid accessor.

Message Syntax

The following is the syntax for a \$Valid accessor:

SET Var=\$Valid(Object.Property,Value)

SET Var=\$Valid(Object.Property.Value(Parameters),Value)

The following example illustrates how the \$Valid function would validate a Social Security Number. First an instance of Employee is created and bound to the

T%Employee temporary variable. Next, the SSN Value property of the Employee object is accessed within the context of the **\$Valid** function. The **\$Valid** function returns a truth value if the SSN is valid. If it is invalid, it returns false and a message is produced.

```
CREATE T%Employee=Framework$Employee
IF '$Valid(T%Employee.SSN,"123-45-5678")DO $Env.Assert("SSN Invalid - Try
again!") Q
```

The following code implements the **\$Valid** accessor.

```
Input:(T%SSN) ;Value passed in from $Valid function.
;If the SSN is not in external form, then return null.
IF T%SSN'?3N1"-"2N1"-"4N Q 0
;Else return true
Q 1
```

The **\$Valid** function requires the first parameter be a System parameter. It is the value to be validated. Optional parameters are permitted. Rules 1, 2 and 3 are applicable.

Input Syntax

Input:(Value, [P₂-P_n,P_{n+1}...])

Where:

Value is the value to validate.

It returns true if valid, false if not valid.

Using Events

Definition of Events

Event handling in EsiObjects is based upon one object (Object A) taking out a watch on another object (Object B) for a particular state change using the **WATCH** command. The watch initiated by Object A sets up a linkage between Object B and itself. It specifies a callback entry point for processing the event. Additionally, the watching object can specify what changes in Object B's state it wants to detect.

At any point Object B can fire an event using the **EVENT** command. The event firing may be a result of a state change or for any other reason. At any point Object B may fire an event. Whenever an event is fired, all objects watching the specific event fired will be called at the specified callback point.

When Object A no longer wants to be notified the event, it issues a command to terminate the event linkage with Object B. The **IGNORE** command breaks the link. Also, if either object dies, the event linkage is broken.

See also the topic [How to Integrate Objects](#) in this guide for information on using events to integrate objects.

The Event Cycle

Assume that Object A takes out a watch on an object for a particular event. The watch specifies what method and label within the method is to be called when the watched event is fired. The watch command is used to establish the watch.

When Object B fires the event, Object A is notified by calling of the method and label specified by Object A. The **EVENT** command sends the notification and causes the callback to be invoked.

When Object A no longer wants to be notified of the event, it issues a command to terminate the event watch. The **IGNORE** command is used to do this.

Also, if either object dies, the event watch is terminated.

How an Object Watches for Events

An object watches another object for an event by using the **WATCH** command. The Watch command is used to set up itself to receive a callback to an entry point when an event occurs. The command specifies the object and event or property being watched, and the entry point of the callback code.

The following is the syntax of the **WATCH** command:

WATCH object.eventname:label^methodname

WATCH object.propertyname:label^methodname

WATCH object.\$EVENTS:label^methodname

WATCH object.\$PROPERTIES:label^methodname

The first two are the syntax for watching for a specific event and property. The last two are the syntax for watching for all events and all properties.

For more information about the WATCH command, see the [EsiObjects Language Reference Guide](#).

Examples

The following example watches an object for a PatientName event:

```
Watch I%NewPat.PatientName:PatName^Events
```

The following example shows the watching of an object for any event that occurs in the object. Note that the callback entrypoint specifies a label name, with no method. In this case, the label must exist in the method where the **WATCH** command is issued.

```
Watch I%NewPat.$EVENTS:PatMod
```

The Callback Entrypoint

The callback entrypoint label specified on Watch commands must exist in the appropriate method and must accept the correct number of parameters that the event notification will pass.

An event notification will pass two (2) parameters plus any parameters passed on the **EVENT** command.

A property notification will pass five (5) parameters.

For more information about the parameters passed on the event notification, see the **WATCH** command in the [EsiObjects Language Reference Guide](#).

How Events are Triggered

An object sends out a notification of the event by using the **EVENT** command.

The following is the syntax of the **EVENT** command:

EVENT eventname

EVENT eventname(parameters)

In addition, any property that is modified via the Assign or Kill accessor will automatically generate an event notification.

For more information about the **EVENT** command, see the [EsiObjects Language Reference Guide](#).

Examples

The following example notifies the system of a PatientChange event:

```
Event PatientChange
```

The following example shows the notification of an event, and the passing of relevant information on the notification:

```
IF I%PatientName'=T%NewName DO
. SET T%OldName=I%PatientName
. SET I%PatientName=T%NewName
. EVENT PatientName(T%OldName, T%NewName)
```

How the Event Notification is Terminated

The **IGNORE** command is used to break the linkage between an object and the object(s) it is watching. The object that issued the **WATCH** command is the one to issue the **IGNORE** command.

Also, if either object in the relationship dies, the event linkage is terminated automatically.

The following is the syntax for invoking a method with the **Ignore** command:

To break all event relationships with all objects:

IGNORE

To break all event relationships with an object:

IGNORE Object

To break a linkage with an object for a specific event:

IGNORE Object.eventname

To break a linkage with an object for a specific property:

IGNORE Object.propertyname

To break a linkage with an object for all events:

IGNORE Object.\$EVENTS

To break a linkage with an object for all properties:

IGNORE Object.\$PROPERTIES

For more information about the IGNORE command, see the EsiObjects Language Reference Guide.

Examples

The following example terminates the all event linkages with an object:

```
Ignore I%NewPat
```

The following example terminates the event linkage with an object for the PatientName event:

```
Ignore I%NewPat.PatientName
```

Using Relationships

Creating and Destroying Objects

Object Life Cycle

The following are the phases in the life of an object:

- **Object creation**
- **Object lifetime**
- **Object destruction**

The phases of an object's life cycle are sequential and are described as follows:

- During object creation, the object is created and initialized.
- During the object's lifetime, the following occurs:
 - Message processing responds to requests.
 - The object's state is restored from or saved to an archive or file.
 - The state of the object is validated.
 - Security for object requests is verified.
- During object destruction, the following occurs:
 - Verification that all users are done with the object.
 - Clean up of any relationships.

For more details about the object life cycle, see [Guidelines for Using Objects](#).

Object Creation

EsiObjects offers you a flexible approach to creating object through the CREATE command. It initializes the object according to keywords on the command. Additionally, it provides for object referencing to insure that the object is not destroyed before it has completed its usefulness to other objects.

CREATE Command

Use the **CREATE** command syntax to create an object. Refer to the Command section of the [Language Reference Guide](#) for a full syntactic description of the CREATE command.

Examples

The following example shows how to create an Address object in its default state. The keyword Shared=1 insures that the instance is persistent and available to other objects for sharing. The keyword Child=1 insures that the object it is a child of the object issuing the Create. The Base="^PATIENT" keyword changes the storage location of the instance. All instances will be created under this global root.

```
CREATE I%CustAddr=Framework$Address:(Share=1,Child=1,Base="^PATIENT")
```

The following example shows how to create an object using creation keywords.

```
CREATE I%MyFile=File:(Share=1)
```

Using the CREATE Method

The CREATE command contains many generic capabilities that are common to the creation of all objects. Often, however, you may want to add functionality to the CREATE command, specializing the creation of an object. The CREATE method performs this function.

The CREATE method resides in the Factory interface of the class. The name must be defined as uppercase. If it is defined, it will be automatically invoked at object creation time.

If the CREATE command contains parameters, these parameters are passed to the CREATE method of the class.

The following example creates an Address object and passes the positional parameters to the CREATE method of the class.

```
CREATE T%CustAddr=Framework$Address("Boston", "MA"):(Shared=1)
```

The CREATE method could look like the following example.

```
Input:(T%City,T%State)
  S I%City=T%City
  S I%State=T%State
  Q
```

Creating Child Objects

Objects are related to each other in one of the following ways:

- Using Relationship
- Containing Relationship

In both relationships, objects communicate with each other by sending messages.

A **Using Relationship** occurs when two objects are related externally. In other words, each object exists apart from the other. An example is a server object that receives requests from external objects and processes their requests. The collaborating objects are independent of each other. The server object defines an interface that the client object messages to invoke its services. The objects, being independent of each other, have independent lives.

A **Containing Relationship** occurs when an object is part of (in other words, is contained in) the internal state of another object. An example is a Customer object (parent) that contains, as part of its internal state, an Address object (child). The child object's lifecycle is closely bound to the parent's lifecycle, specifically; the child will be destroyed when the parent is destroyed.

Creating Private Objects

An object is created as a private object by default. This means that only the creating context can access the object unless it explicitly passes the reference to another object. Also, once the creating context is destroyed, any objects created within the creating context are also removed from the system.

In the following example, a private List object is created.

```
CREATE I%MyList=List:(Share=0)
```

Creating Shared Objects

An object can be created as a shared object by using the Share keyword on the CREATE command. If the object is to be shared outside the current context, then use the Share=1 keyword. In this case, the object survives beyond the scope of the calling context and remains until it is deleted explicitly.

In the following example, a shared List object is created.

```
CREATE I%MyList=List:(Share=1)
```

Object Preservation

Within its lifetime, an object may provide services to other objects. The service object may stay around indefinitely. However, it may often come into being for a short duration but be available to other objects. The concept of object referencing has been implemented in EsiObjects to insure that an object is not inadvertently destroyed before its time.

PRESERVE Command

The **PRESERVE** command is used to increment an objects internal reference count. Refer to the Command section of the [Language Reference Guide](#) for the PRESERVE command syntax.

The **PRESERVE** command should be used in conjunction with the **DESTROY** command. A using object executes the PRESERVE command to increment the internal reference count of an object it is using to insure that the object will not disappear before it has completed its operations. Once finished using the object, the using object can execute the DESTROY command. The DESTROY command decrements the internal reference count. When all user objects have issued the DESTROY command, the next destroy action will delete the object.

Relationship to the DESTROY Command

For every PRESERVE command that is executed on an object, the complementary DESTROY command should also be executed. Essentially, executing these complementary commands insures that the object will not be destroyed within the time duration between their executions.

Object Protection

Often within a working relationship between two objects, one object needs pass its OID (or an OID of an object that it owns) back to a calling object. To protect itself or the object it owns from being preserved or destroyed, the object OID should be passed back via the \$PROTECT function (See the \$PROTECT function within the Functions section of the Language Reference Guide). The \$PROTECT function modifies the OID such that the DESTROY and PRESERVE commands have no effect when applied.

Object Destruction

DESTROY Command

To destroy an object with the **DESTROY** command means to remove it from the system. Refer to the Command section of the [Language Reference Guide](#) for the **DESTROY** command syntax.

The **DESTROY** command accepts a reference to an object or any expression that evaluates to an object reference.

The following is an example of how to destroy an Address object:

```
DESTROY I%CustAddr
```

Protecting an Object from Destruction

There are two ways of protecting an object from destruction.

The first way is to protect it from being destroyed while being used by other objects. Use of the **PRESERVE** command permits the object to defer destruction by the **DESTROY** command until the last **DESTROY** is executed, at which time the object will finally be destroyed (See the **PRESERVE** command in the Language Reference Guide).

The second approach protects the objects from being destroyed at all times. Using the **\$PROTECT** command to change the status of the objects OID will protect the object from being destroyed. Applying the **DESTROY** command will have no effect.

Using the DESTROY Method

The **DESTROY** command is used to delete an object. Sometimes it is important to add functionality to the destruction process. The **DESTROY** method specialize the destruction of an object.

The **DESTROY** method resides in the Factory interface of the class. The name must be defined as uppercase. If it is defined, it will be automatically invoked at object destruction time.

The following example illustrates how to destroy an object. Note that if the object bound to the I%CustAddr instance variable is destroyed, the variable I%CustAddr will remain defined. The variable must be removed via the **KILL** command.

```
DESTROY I%CustAddr
```

If the **DESTROY** method exists in the Factory interface it will be executed. The **DESTROY** method can be used to terminate the destroy operation based on some internal or external condition as illustrated below. Assume that the Address object bound to the I%CustAddr instance variable contains an I%RefCount variable that contains a count of the number of customer objects using it. If the count is greater than 1, then the Address object must continue to live. However, if the count is less than or equal to 1, it can be destroyed. Issuing the **QUIT** command with a value of 0 (same as setting **\$RETURN=0**) tells the **DESTROY** command not to destroy the object. Returning with a 1 (same as setting **\$RETURN=1**) tells the **DESTROY** command to destroy the object.

```

; If Address object used by another object, decrement the reference count
; and return false to abort the destroy.
IF I%RefCount>1 SET I%RefCount=I%RefCount-1 QUIT 0
;Else, quit with true - object will be destroyed.
QUIT 1

```

Testing to See If an Object Has Died

If the **DESTROY** command results in an dead object, then **\$TEST** is set to 1. The **\$TEST** special variable is 0 if the object still exists. Note that the **DESTROY** command releases any interest the current context has in the specific object. However, the object

may decide to continue to exist because other contexts may still be using the object. If the **DESTROY** command is successful, subsequent reference to the destroyed object will result in an error. If references are made to the object, you should always test for its existence before making the reference.

In the following example, the object attached to the I%CustAddr variable is issued a destroy command. The results of the destroy action are stored in the special variable \$Test. If the object was in fact destroyed, \$Test will return 1. If the object was not destroyed, it will be set to 0. In the example below the IF command tests \$Test, if true the Event command is used to fire the AddressDied event.

```
DESTROY I%CustAddr IF $TEST Event AddressDied
```

When outside the context of the DESTROY command, you may use the **\$EXIST** function to test if an object exists. Refer to the Functions section of the [Language Reference Guide](#) for the \$EXIST syntax.

If the object exists, the function returns 1. If the object does not exist it returns 0. The example below tests for the existences of the customer address object, that is bound to the I%CustAddr instance variable. If it does not exist, a dialog box is asserted with a message.

```
I '$EXIST(I%CustAddr) DO $Env.Assert("Customer Address does not exist.")
```

Additionally, you may use the **\$INFO** to test the existence of an object. Refer to the Function section of the [Language Reference Guide](#) for the \$INFO syntax.

The following example accomplishes the same results as the **\$EXIST** example above.

```
I '$INFO(I%CustAddr,3) DO $Env.Assert("Customer Address does not exist.")
```

Using Class Libraries

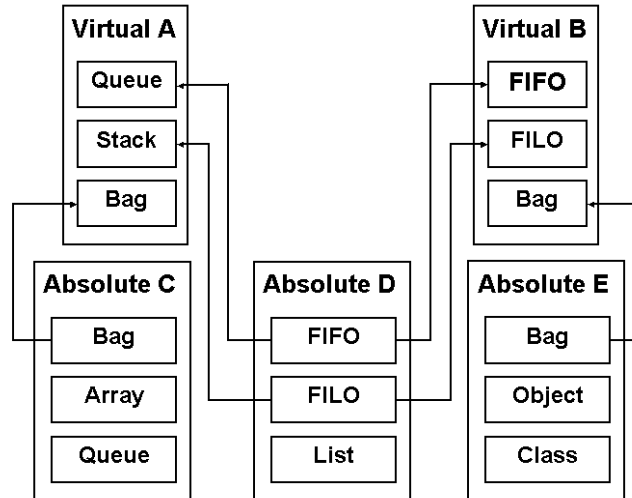
EsiObjects supports Class Libraries. Libraries are used to group classes by some artificial criteria that is usually based on application or organizational requirements. Libraries provide a firewall between groups that prevent inadvertent damage to protected classes.

Two types of libraries are supported:

- Absolute
- Virtual

Absolute libraries physically contain classes. **Virtual** libraries do not physically contain classes; instead, they can integrate classes contained in one or more absolute libraries. A virtual library can also view some classes in its absolute libraries by alternate names. It can even have multiple entries for a single class, each under a different name.

The relationship between virtual and absolute libraries is hierarchical, and never more than one level deep. A virtual library imports classes from at least one absolute library, but there is no restriction on the number of classes it can import, and the number of absolute libraries from which they can come. Some or all of the classes in an absolute library may be exported to any virtual library. An absolute library can export its class names to many different virtual libraries. A virtual library is so flexible that it can view any combination of classes in any absolute libraries by any valid names. The following diagram illustrates these concepts.



In this figure, there are two virtual and three absolute libraries. *Absolute Library C* exports its **Bag** class to *Virtual Library A*, while *Absolute Library E* exports its **Bag** class to *Virtual Library B*. This causes no conflict, but if *Virtual Library A* wants to import *Absolute Library E*'s copy of **Bag**, it will have to do so under a different name.

Absolute Library D exports its **FIFO** and **FILO** classes to *Virtual Library B* under the same names. It also exports them to *Virtual Library A* under different names: **FIFO** as **Queue**, and **FILO** as **Stack**. The same class **FILO** can be viewed by one virtual library as **FILO** and by another as **Stack**. Any number of virtual libraries under any combination of names can view it.

When creating objects with the **CREATE** command, the class name specified is relative to the current default library. By default, if you do not specify the library name along with the class name in the **CREATE** command; the master virtual library is used to resolve the requested class name. Classes should always be referred to using a full name that includes their library name to avoid any confusion or programming errors. This is due to the fact that the same class name can be used in different libraries. Therefore, adding the library name to the class makes it a unique reference since two libraries with the same name cannot exist.

The format of a full name is as follows:

LibraryName\$ClassName

This form of name can be specified anywhere that a class name can be used. The following is an example of a full class name:

Base\$List

where:

Base is the library name

List is the class name

Absolute and Virtual Libraries

The relationship between virtual and absolute libraries is hierarchical (never more than one level deep). A virtual library needs at least one absolute library, but it can have more than one.

Absolute libraries physically contain classes; so all the fields on the Library Browser have meaning for an absolute library. Virtual libraries do not physically contain classes but can integrate classes contained in one or more absolute libraries. A virtual library can also view some classes in its absolute libraries by alternate names. It can even have multiple entries for a single class, each under a different name.

A **virtual library** imports classes from at least one absolute library. There is no restriction on the number of classes a virtual library can import, and the number of absolute libraries from which virtual libraries can come.

Some or all of the classes in an absolute library can be exported to any virtual library. An absolute library can export its class names to many different virtual libraries. A virtual library is so flexible that it can view any combination of classes in any absolute libraries by any valid names.

Using Virtual Libraries

By default, if a simple class name is used, then the current default virtual library is used to determine the actual class. It is also possible to reference a virtual library explicitly by name.

In the following example, a List object is created using the default mechanism and using an explicit reference to the virtual library called Master. The absolute library that owns the class, which creates this list, is hidden.

```
CREATE I%MyList1=List  
CREATE I%MyList2=Master$List
```

Using Absolute Libraries

In most cases it can be required that an object come from a specific absolute library. When this is the case, using an explicit reference to the source library can create the object. All references to a class should be explicit.

In the following example a List object is created from the ESI library:

```
CREATE I%MyList3=Base$List
```

Integrating Objects

Elements of Integration

Object Contracts

Object contracts are specific interactions and responsibilities that are expected of an object. Generally a contract can be viewed as the operations and events that a specific object must support. The object makes a commitment to provide services that calling objects can use.

Contracts are also used to describe relationships. Each party in a relationship is expected to fulfill a specific contract. Only those objects that fully comply with the contract associated with a relationship can be used in that relationship. Depending on how tightly objects are integrated, the extent of their contract varies.

An example of a simple contract is an object that guarantees that an event is not fired until a specific condition occurs. For example, the basic contract for a person object is that it fires a property event only when one of its properties has been modified or deleted.

Object Responsibilities

The responsibilities of an object are defined as the aggregate of all the contracts that the object must fulfill. For example, a Person object in a healthcare system has a number of different responsibilities, some of which include the following:

- Maintain the integrity of its state data.
- Manage relationships with other objects it is collaborating with.
- Make known any changes in state that other objects should know about.
- Protect itself from inadvertent destruction or harm.

The responsibilities of an object convey a sense of purpose for an object and help place an object within the system.

How to Integrate Objects

Grouping Integration

Maintaining Groups

Objects can be integrated by forming groups of objects that obey the same contract.

The following are ways to establish a grouping relationship:

- Common class
- Common interface
- Key property

Objects that share a common class inherit the operations from that class. Therefore, these objects' responsibilities include the specific operations defined within their owner class and the operations inherited by the shared class.

The following EsiObjects collection classes all share the same Collection class:

- Set
- Bag
- Dictionary
- Log
- Array
- List
- Map
- MultiMap

Behavior implemented in Collection is inherited by the subclasses and are understood by objects of those classes. Although the specific subclass can override the implementation of the behavior, all Collection objects understand the shared operation. For example, the Collection class defines an operation called `InsertElement`. All classes that share this class implement and understand this operation.

Objects can also share a common interface. An example of this is the EsiObjects class `ABS_SECURITY`. This is a abstract class that contains one interface called `Security`. This class can be used as a mix-in class.

A mix-in class is a detached, standalone class that is used solely as a superclass for other classes that want to inherit the operations or data defined in the mix-in. `ABS_SECURITY` then can be mixed in with other classes. These classes share the `Security` interface and all objects created from these classes include the components defined in that interface. Any objects that need to implement object security can make use of this interface. Therefore, dissimilar objects can integrate the common functionality defined in the shared interface.

Applying Operations Across a Group

When a group of objects obey the same contract, you can apply operations across the group by using any of the following methods:

- Using an **iterator**
- Using **ForEach**
- Using a **callback**

If a group of objects are placed in a collection, you can use an iterator to access each member in the collection and apply the operation to each object.

Event Based Integration

How Events Work

Event-based integration is the most common form of object integration. This form of integration is used when the state of one object needs to be updated when the state of another object is changed.

EsiObjects contains 3 commands that implement event based integration of objects. An event is generated with the **EVENT** command. Event watches are controlled with the **WATCH** and the **IGNORE** commands. For more information about the **EVENT**, **WATCH** and **IGNORE** commands, see the [EsiObjects Language Reference Guide](#).

Referring to the diagram above, the Subject is the object that is being watched. The Subject determines under what conditions an event is fired. A Subject can have more than one Observer.

The Observer is the object that is watching; it establishes and breaks an event watch. The Observer decides what objects and events to watch. Any object can be an Observer and a Subject.

Events are attached to an Event Handler in the Observer object. An Event Handler is a body of code associated with the Observer and is invoked when any Event is fired. It has an entry point and accepts parameters. The event handler must accept the parameters passed in to handle the event properly.

There are **three** basic types of event handlers:

1. Specific to the object and an event
2. Specific to an event
3. Generic

Setting Up an Event Watch

When doing model (server) side programming, an object is often required to publish events that are of interest to other objects. As stated above, EsiObjects has three commands that offer event-handling services.

There are **five** general steps to publishing an event and subscribing to it. They are:

1. Determine what event to watch.
2. When the event fires, where will it go. An entry point and parameter list must be defined.
3. Write the code to handle the event.
4. Determine where to establish the event watches.

5. Determine where the watches are to be broken.

To illustrate how to set up a manual event watch, let's apply the steps using a specific example. Assume that the central computer in a hospital monitors all fire alarms. Smoke detectors in each room generate an event if smoke is detected. A Floor object watches the event. The Floor object evaluates the detection and triggers a selected event, which supplements the original event with the floor number. Also, the floor alarm system is activated. The hospital watches every floor to see if there are other emergency events and handles them appropriately.

The following steps illustrate how the Fire detection of the Hospital Fire Detection application was set up using the steps listed above. Use the Session Browser to display the event and method code.

1. Using the Session Browser, a new event was defined within the Floor class called Smoke. This event definition contains a code stub that shows the parameters required by the actual event handler.
2. Next, the location of the event handler had to be defined. In this case, it resides in the FireWatch method of the Floor object. This is the same method that is used to set up an event watch. Upon start-up, this method is invoked by the ActivateEmergencySystems method of the Hospital for each room in the hospital. The Watch command is executed which sets up an event watch for each room.
3. The actual event handler Smoke is stored in the FireWatch method of Floor with a public label. The handler is written as a procedure, therefore it does not return a value. In this case, the Smoke handler is specific to the Floor. If executed, it triggers another event Smoke that is stored in the method FireWatch of the Hospital class.
4. Next the event watches must be located. In this case, the watch is located in the FireWatch methods of Floor and Hospital. Refer the EsiObjects Language Reference Guide to determine the format of the **WATCH** command.
5. Finally, the watches must be turned off. In the case of the Hospital Fire Detection application, the DeactivateEmergencySystems method of the Hospital class would be one place to perform this function. Use the **IGNORE** command to ignore the event when you are done with the object. Refer the [EsiObjects Language Reference Guide](#) to determine the format of the **IGNORE** command.

Destroying an object will force ignores of event watches on the object.

Watching Properties

It is possible to watch for changes to the properties of an object. Currently in EsiObjects, the only way to watch properties is to set up the watch manually as outlined in the last section.

The **WATCH** command uses the property name or **\$PROPERTIES** keyword in lieu of an event name. The **\$PROPERTIES** keyword specifies that all properties are to be

watched and any change (a set or a kill) to any property in the object will trigger the callback.

Event Handler

The **WATCH** command specifies the object to watch, the event or property name to watch within the object, and the callback label and method to be invoked when the event is fired. This callback label is also known as the **Event Handler**. This label is created in the method specified on the **WATCH** command and must accept at least 2 parameters for any event watch and 4 parameters for any property watch. See the [WATCH command](#) in the [EsiObjects Language Reference Guide](#) for details on the watch command and the callback parameters.

The callback label you write must accept the minimum parameters plus any parameters that the event may pass itself.

If the **WATCH** command specifies a label for the callback only (no method is specified) then the label must be created within the method where the **WATCH** command is invoked. Additionally, the label must have the **Public** label keyword as the following example shows:

```

Watch T%Patient.$PROPERTIES:PatLab
QUIT
; callback label for patient watch
(Public)PatLab(Obj,Prop,Callfram,Oper,Value)
; property watches must accept five parameters
;callback code goes here
QUIT

```

If the **WATCH** command specifies a label and a method, the syntax for the callback label is the same as above, except the **Public** keyword is not needed.

Creating a Generalized Event Handler

General Event Handlers

It is possible to establish a watch on all events and properties associated with an object. You can use the following event names to create a generalized watch on events and properties:

- **\$EVENTS**
- **\$PROPERTIES**

When you use these event names, the event handler must be able to accept all possible input.

Under some circumstances the entire state of an object can change. When this occurs, the generalized handler for a property receives a special event named **\$PROPERTIES** to indicate that the entire state has changed. When the **\$PROPERTIES** event is sent, it is sent alone. The general handler does not receive a call for each property.

The following example shows a generalized property change event handler.

```

Props(Object,Property,Callframe,Oper,Value) ;Generalized property handler
;
;Update value of the property in the property list object
DO I%PropList.Set(Prop:L%Property,Value:L%Value)
QUIT

```

The following example shows a generalized event handler.

```
Event(Object,Event,P1,P2,P3,P4,P5,P6,P7,P8,P9,P10,P11,P12)
; Generalized event handler
;
; This generalized handler causes the event to be echoed
; to anyone watching this object
;
; Determine the number of parameters
FOR A=1:1:13 QUIT:'$DATA(@"P"_A)
IF A=1 Event @Event@
IF A=2 Event @Event@(P1)
IF A=3 Event @Event@(P1,P2)
.
.
.
IF A=12 Event @Event@(P1,P2,P3,P4,P5,P6,P7,P8,P9,P10,P11,P12)
QUIT
```

Establishing a Generalized Event Watch

The following is an example of watching all events for an object:

```
; Watch all events, invoke Events label when any event occurs
WATCH Object.$EVENTS:Events
```

The following is an example of watching all property changes for an object:

```
; Watch for any property change, invoke the Props Label
; if the property changes
WATCH Object.$PROPERTIES:Props
```

Dissolving a Generalized Event Watch

You can use the **IGNORE** command to dissolve a generalized event watch. Use the event name **\$EVENTS** for events and use the property name **\$PROPERTIES** for properties. This dissolves the generalized event watch in the requested object. Any specific events and properties being watched continue to be in effect.

Generalized event watches also are dissolved with the argumentless **IGNORE** command, and the **IGNORE** command specifying just the object.

The following example shows how to dissolve a generalized event watch.

```
WATCH Object.$EVENTS:Events ; Watch all events
WATCH Object.Name :Name ; Watch the name properties
WATCH Object.$PROPERTIES:Props ; Watch for any property change
.
.
.
Ignore Object.$EVENTS ; Ignore the generalized events
Ignore Object.$PROPERTIES ; Ignore the generalized properties
; At this point still watching the name property
```

Relationship Integration

Object Relationships

Within complicated applications, objects frequently need to collaborate with other objects to accomplish tasks they are responsible for. The behaviors that make up a system are realized by objects that collaborate with each other. The concept of a contract between objects was discussed in [Object Contracts](#). The relationship between any two objects encompasses the operations that can be performed between the two, and the behavior that results. For example, a Patient object enters a relationship with a Provider object in a medical application.

Objects can be integrated with each other via a **Hierarchical Relationship** where a parent object is made up of one or more child objects. These child objects in turn can be parent objects to one or more objects of which they are made. These hierarchical structures are often referred to as Composites.

Besides a hierarchical relationship, objects can be related via an **Owning Relationship** where one object owns the objects it is related to. The owned object is not a child object, in that it does not exist within the internal state of the owner object. Instead, the object can receive messages and can be involved in other relationships with other objects apart from the owner object. However, the owner object maintains the objects, perhaps creating and destroying them. In any case, the owner object can exist without the objects it contains, but the contained objects cannot exist without the owner.

One other type of relationship between objects is a **Using Relationship**. This is where an object can use another object, but each object exists as a peer object of the other. Neither object owns or contains the other.

Parentage

In a hierarchical relationship, parentage is defined and maintained by the parent object. The child objects are defined as part of the parent object's internal state. The parent object when created can automatically create the child objects or can create them as needed. Objects that are parents are responsible for the destruction of their children

Mediation of Requests

In a hierarchical or owning relationship, an object receives a message, and if necessary delegates the message to the appropriate object. This object can be a child object or a parent object if the object is a child object.

For objects within dissimilar systems that do not know how to communicate with one another, some mediation must be provided to allow communication to occur. For this purpose, any object wanting to be advised by another object must support the OnAdvise operation. The object being watched must support the Subscribe and DeSubscribe operations. The concerned object can invoke the Subscribe operation on the target object to be

advised of changes within the object. This relationship is closed when the concerned object invokes the DeSubscribe operation.

Lifetime Issues

When using an object in a relationship, it is essential to ensure that the object used has a sufficient life span to be used in the relationship. The general rule for determining relationships is that the object used in the relationship should have a life span of at least that of the objects to which it relates.

Guidelines for Using Objects

Object Life Cycle

It is important to understand the life cycle of an object to understand what can be done with an object. There are three major phases in an object's life cycle with which you need to be concerned:

- **Object creation** - The phase in which an object's resources are allocated and the initial state of the object is defined.
- **Object lifetime** - The phase in which the reaction to specific object requests, event monitoring, and triggering occurs.
- **Object destruction** - The phase in which determining appropriateness, breaking relationships, destroying component objects, and deallocating resources occurs.

An implicit contract is made when you create an object. This contract states that for every object you create, you must either destroy the object or transfer the responsibility of destruction to another party (object). Keep in mind that it is possible to have a pointer to an object that no longer exists.

The following sections describe the phases of the object life cycle in more detail and use an example class hierarchy that models a hotel. All classes used in the example can be found as descendants of the class Examples.

Object Creation

An object is created with the **CREATE** command. The syntax is as follows:

```
CREATE I%Obj=Classname(P1=Value,P2=Value):(Keyword=Value):(Prop1=Value,Prop2=Value)
```

The steps the EsiObjects system will following when creating an object are listed below:

Object allocation When you use a class name and creation parameters to create an object, the system allocates space for the object in the appropriate area. The system also allocates an Object ID (OID) for the new object and assigns the object to the new OID. After this is done, the variable is assigned the OID of the new object. The internal reference count is set to 1.

CREATE method Once the object has been created as described in step 1, the **CREATE** method for the object is run. This method is invoked with the **CREATE** method parameters. Note the following about the **CREATE** method:

- a. Defines the initial default state of the object.

- b. Establishes the basic relationships for the object. Therefore, any immutable relationships are often passed as CREATE method parameters.
- d. Runs like any other method.
- e. Normally chain calls their ancestors.

In a formal sense, the CREATE method is named `Factory::CREATE`, which indicates that the CREATE method is a part of the factory interface.

Note: Do not invoke the `Factory::CREATE` method directly.

Property assignments

Once the CREATE method is executed, then any property assignments are applied. All assignments occur from left to right. Errors during assignment result in a run-time warning. Any additional property assignments that occur after an error are processed.

When assigning a property using the **CREATE** command, the Creation accessor is used. If the object has no Creation accessor, then the Assign accessor is used. Attempts to assign properties that do not have a Creation accessor or an Assign accessor result in a run-time warning.

*Note: Some properties can be assigned only at the time of object creation (properties that have a Creation accessor but no Assign accessor). It is possible that the Creation accessor can cause the assignment of the properties to occur during its processing (by using the **ZAPPLY** command).*

Object Lifetime

An object begins its life once it is created. During the course of an object's life it can undergo many changes. Each of these changes is the result of the object interacting with other objects, interacting with the environment, and with the user. All interactions are based on the following mechanisms:

- **Properties**
- **Events**
- **Methods (or messages)**

These mechanisms are discussed in the following sections.

Properties

Properties Defined

Properties are the parts of an object that are seen by the outside world. Properties are the way that you can distinguish one object from another. Each property is a logical sense of some part of what the object is. The actual physical state of the object may not reflect the property directly, but is instead used to synthesize a value when one is requested. The opposite is also true. When a user assigns the property of an object, it might entail more than a simple internal state change.

A Person object can have properties, for example:

- **Name**
- **Date Of Birth**
- **Social Security Number**

Property Assignment

Once an object has been created, it is possible that some of its properties can change over time. Many are assigned after the object has been created. For example, collection iterators allow the `IterationOrder` property to be changed at any time.

When a property is assigned, the value assigned to it may have no relationship to what the property value is on lookup. This is because property assignment logic validates and transforms the input value.

The following accessors are used in property assignment:

- **Assign** - Assigns a value after creation.
- **Creation** - Determines the initial state.
- **\$Valid** - Checks the validity of an assignment.
- **\$Normalize** - Changes the value from an external format to an internal format.
- **Kill** - Resets the collection to the default state or removes an element from a collection.

Some properties support the **\$VALID** function, which can be used to determine if it is valid to assign a certain value to a property. The **\$VALID** function uses the **\$Valid** accessor for the requested property. Attempting to assign a property that does not have an **Assign** accessor results in a run-time error.

It is important to provide an **Assign** accessor when you provide a **\$Valid** accessor. If an **Assign** accessor is not provided, attempts to claim that a value is a valid assignment results in an error.

Property Lookup

A property lookup is a request for some piece of information about an object. Generally this is some characteristic of the object that is of concern to the object's user. The most common form of property lookup is the Value accessor, which is used to find out to what a property is set. The Value accessor returns the current object state in a normalized form (in other words, suitable for a user and an Assign accessor).

The following accessors are used in property lookup:

- **Value** - Finds the value of a property.
- **\$Get** - Finds the value if there is a value. If there is no value, the default is normalized.
- **\$Order** - Checks for the next element in a collection property.
- **\$Data** - Checks if an element exists in a collection property.

Message Processing

Types of Message

The following is a list of the types of messages supported in EsiObjects:

- **Inquiry** — request for information
- **Modification** — request for a change in object state
- **Instruction** — request to perform a task

Inquiry Messages

An inquiry message asks the object something about itself. Messages that ask about identity, class, protocol membership, and current internal state are inquiry messages. An example of an inquiry message is a message that asks a collection how many elements it contains.

Modification Messages

A modification message tells the object to modify its current internal state. An example of a modification message is one that tells a collection to remove one of its elements.

Instruction Messages

An instruction message tells the object to perform some task. This often results in multiple messages being sent to other objects, which might not have been visible to the original sender. The **ForEach** message is an example of this kind of message because it asks a collection to send some message to each one of its elements.

Generating and Processing Events

Event Commands

Event generation and event processing can play a major role in the life cycle of an object. Every change to the state of an object is the result of some event (usually a user action). Events are not limited to user's interface objects. Events can be of even more use when used in modeling the applications domain.

The main advantage of events is their loosely bound nature. Events provide a mechanism that allows one object to cause other objects to react to a change. There is no need for the event generating object to know or care about what objects are receiving the event. In EsiObjects, the following commands control events:

- **WATCH**
- **IGNORE**
- **EVENT**

For more information about the EVENT, WATCH and IGNORE commands, see the EsiObjects Language Reference Guide.

Event Generation

The events generated by an object are of two major forms:

- Generic Events
- Property Change Notifications

The **EVENT** command can be used to generate both types of events. Property Changed Notifications also occur automatically when a property is assigned or killed. Generally an event is a notification that something has happened versus an advisement that something is going to happen.

Generally, the events that are generated by an object can be determined by examining the interface of the object. The events are listed in the interface. Also, a response template is provided with each event to describe the parameters passed with the event and the general intent of any event handler. The nature of the **EVENT** command is such that it is possible for an object to generate events that are not in its interface. However, this behavior is strongly discouraged.

Event Response

Event handlers are methods that have been designed to respond to specific events. The event handler is task oriented and does not directly support returning any information as a result of its invocation. The association of an event handler to an event and an object is made using the **WATCH** command.

An object can choose to watch for any event (within the bounds of security). A warning is generated if it is believed that the object does not actually generate the event.

Event watches can be turned off by using the **IGNORE** command.

Event Delivery

The **EVENT** command generates an event. When this command returns, the event has been queued, but necessarily has not been delivered. This means that you cannot use events as a blocking mechanism on an action.

EsiObjects often batches the delivery of events to a specific object. Therefore, object A and object B receive events. It is possible that object A handles all of its events prior to object B handling any events.

Events always are placed in the event queue in a first-in /first-out (FIFO) basis. This applies to a receiving object. Events always are delivered to a specific object in the order they were posted to the object. Note the following:

- Events always are delivered in order on a per object basis.
- The timing of events is not guaranteed.
- Events are dequeued by the top-level event loop and the **EVENT** command.

Object Destruction

DESTROY Command

The life cycle of an object comes to an end when it acknowledges a **DESTROY** command. An object's lifetime also can end if it is a child of another object that is destroyed. Once an object is destroyed, its pointer becomes invalid.

The destruction of an object involves three major phases:

- **DESTROY command**
- **DESTROY method**
- **Deallocation**

Note that objects are destroyed with the **DESTROY** command or are destroyed by being in a relationship that causes their destruction. It is possible to lose objects that are not properly destroyed (this happens when the last pointer to an object is killed).

Caution must be used when using scoped variables or the **KILL** command to ensure that any object the variables are pointing at gets cleaned up properly. Unshared objects that are lost are automatically deallocated at process rundown.

The steps for destroying an object are described below:

DESTROY command

The **DESTROY** command is invoked with the object as its argument. The **DESTROY** command attempts to identify the requested object. If it is not a valid object, the request is ignored. First the internal reference count is decremented by 1. If it is greater than zero, the destroy process is terminated at this point. Next, the **DESTROY** method is invoked (see step 2) if it exists. If this method returns true, then the **DESTROY** command proceeds with deallocation. The **\$TEST** special variable is set to true if the expression is no longer a valid object. This means that true is returned if any of the following are true:

- The request did not point to a object.
- The request has a dead object.
- The object had been destroyed.

DESTROY method The **DESTROY** method is invoked on an object to allow it to prevent its destruction and to allow it to perform the required cleanup. The typical **DESTROY** method consists of four parts:

Feasibility Determines whether an object should be destroyed. If the object is not to be destroyed, **\$RETURN** is set to false and the method is exited. The feasibility is often determined by checking a reference count, or by confirming the request with relationships.

Clearing relations Objects can form relationships with other objects. When an object is destroyed, it is necessary to break these relationships. There are two types of relationships:

Pointer-based relationships use a protocol for maintenance and require individual treatment.

Event-based relationships are maintained by the system and can be destroyed by the system (via the **IGNORE** command).

Any relationships that have been formed by the object need to be broken. This allows the other object to remove invalid object pointers and might also cause the related object to decide that it should cease to exist.

You must destroy any variables that have been used to track relationships. This prevents the system from having to determine if the relational object should be destroyed.

Generally it is a good idea to ignore any watches created by the object you want to destroy. This is generally done by using the argumentless **IGNORE** command. This step is optional because the deallocation process removes any watches automatically.

Component destruction Destroy individual components of the object explicitly. This is especially true for those components that are not child objects.

Releasing external resources All external resources related to the object must be released. This includes public structures (such as global and local variables) and private structures (such as external objects and heap memory).

The full name of the destroy method is `Factory::DESTROY`.

Note: Do not call this method directly.

Often the **DESTROY** method chains to its ancestors to ensure that everything is cleaned up. There is a possibility that other users may still attempt to access the object while this method is executing. Objects should be coded to be robust during this time period by using the `EsiObjects` **LOCK** command.

Deallocation

If the **DESTROY** method returns a true result, the **DESTROY** command activates the deallocation process. This consists of the following actions:

Destroying children	Any child object is destroyed, which gives the DESTROY method any opportunity to execute. Currently, child objects are always deallocated, regardless of the return value of the child object's DESTROY method. Note that this process is recursive so that all descendant child objects are destroyed.
Canceling watches	Any watches that were formed by this object or formed on this object are removed.
Killing the instance table	The system removes the instance table structures associated with the object, and invalidates the object. It is at this point that the object can no longer be called.

Effects of Object Destruction

The destruction of an object does not remove all pointers to that object. This means that it is possible to have invalid object pointers. Generally, attempts to use these pointers result in an error.

In the following example, an attempt to use an object that is dead results in an error.

```

;T%Acct = an Account Object
;Try destroying the object
DESTROY T%Account
;Try using the object (might work if the object did not go away)
SET T%Bal=T%Account.Balance

```

Care must be taken when sharing a child object because the child ceases to exist when the parent object ceases to exist.

The **\$EXIST** function can be used to determine if an object pointer is valid. This function is used if you doubt that the object still exists.

You can use messaging keywords to ignore an object's existence when sending a message to it. The keyword to use is `existence`. If the object no longer exists, the message is a void operation and returns a null value.

Adding Interfaces to an Object

For a complete description of the interfaces supported by EsiObjects see the [Object Interface](#) subsection of the [Using Objects](#) section in this guide.

Often, however, the Primary interface is not appropriate for the definition of all object services. In many cases you will need to add services to object that you do not want to expose to the user through the Primary interface. As discussed in the [Using Objects](#)

section, the Factory interface is one specialized interface that is very important in the development of an application. It permits you to group all services that are responsible for the creation, preservation and destruction of an object in one interface.

Assume that you wanted to add a set of services to the object that was responsible for security. Given the requirements of security, you certainly would not want these services exposed to the user.

Adding interfaces to a class can be accomplished in two ways:

1. Using a Mix-In class already defined to add the structure and service templates.
2. Creating an interface through the Library Browser and adding the services as you would an any other interface.

In the spirit of reusability, EsiObjects provides a number of Mix-In classes found in the Base library that provide interface service templates for some of the common functions you would find in an application. They are:

- AbsAttachmentObject
- AbsDebugObject
- AbsLockableObject
- AbsSecurityObject
- AbsSerializationObject

These classes are templates and as such, provide only the abstract definition of the service, that the implementation. That is up to you.

If you find the class you want to use, simply link it to the class you want the interface to appear in. Through the multiple inheritance mechanism supported by EsiObjects, the interface will appear as inherited within your class. At this point you can override the services defined in the interface and implement the code at your class level.

If you do not find the Mix-In that meets your needs, it is a simple task to create a Mix-In. It is like creating any other class. However, you simply change the class property to identify it as a Mix-In.

Object Navigation

One of the basic questions in working with an object is simply how do you get the object. Throughout this guide, a number of different mechanisms have been used for getting to and from objects. The following techniques can be used to get to an object:

- **Creation** - Create the object you want to use.
- **Nested access** - Find the object as a property of another object.
- **Data sources** - Request an object from another object that knows where it is.

- **Events and messages** - Receive an object reference as a parameter on a message or an event.
- **Globals and locals** - Retrieve an object from a publicly accessible table.
- **Named** - Retrieve an object by name within the domain.

Creation

One of the easiest ways to get to an object is to create one. Generally this is done when the object is used in some task, or when the object is being created to form a model. When using this form of access the user takes ownership of the object.

The following example creates a new booking object:

```
CREATE T%Booking=Booking:(Child=0)
```

The following example gets an iterator from a Collection:

```
SET T%Int=T%List.CreateIterator
```

Nesting

Often an object exhibits other objects through a property, which allows the nested object to be revealed to other users. When working with nested objects, the ownership of the object remains unchanged. Note that each nested object must exist to prevent an error.

The following example shows how to access the current booking for a specific floor and room number in the Hospital application:

```
; Finds current booking for room 99 on sixth floor
SET T%Booking=I%Hospital.Floor(6).Room(99).CurrentBooking
```

The following example shows how to access the current booking for any room. Object existence is ignored.

```
; Verifies nesting and if valid gets current booking
;
; T%Flr = The Floor Number
; T%RmNm = The Room Number
;
SET T%Room=I%Hotel.(Existence)Floor(T%Flr).(Existence)Room(T%RmNm)
IF T%Room="" SET T%Booking=T%Room.CurrentBooking
```

Data Sources

Many objects know a number of different objects that they do not directly reveal. Such objects often contain specialized methods for finding a specific object based on some user criteria. These methods generally search for objects using the specified criteria and return objects that match. Generally, the ownership of the found object remains unchanged.

Some data sources provide support for the transfer of ownership to those objects they contain. For those data source requests that return a collections of objects that match the

selected criteria, the ownership of the collection (but not the objects that it contains) become the responsibility of the class.

The following example shows how to find a booking for a specific room on a specific date. The return result is either Null (not found) or the booking for the room. There is no change in the ownership responsibility of the booking.

```

; Purpose: Searches for a reservation, given the full
; room number and time to which it applies
; Returns: the matching reservation or NULL if not found
Input:
(
(Required)FullRoomNumber:P%FRN, ;room #:Floor-Room(10-102)
At:P%At="" ; timestamp for the reservation - default to NOW
)
; parse out the floor and room numbers
SET T%FloorNo=$PIECE(P%FRN,"-")
SET T%RoomNo=$PIECE(P%FRN,"-",2)
;get the floor object from the floor table dictionary
; will return a List of floors with the specified
; floor number. The set should have only 1 entry
;- quit if it does not
SET T%FloorSet=I%FloorTable.RetrieveElementsByKey(T%FloorNo)
IF T%FloorSet.Cardinality'=1 Destroy T%FloorSet QUIT
; get the floor object
SET T%Floor=T%FloorSet.RetrieveFirstElement
Destroy T%FloorSet
; get the room object(s) associated with the
; specified room number
; The Room property will return a list collection
; of the rooms matching the room number -
; there should only be 1 entry - quit if not
SET T%RoomSet=T%Floor.Room(T%RoomNo)
IF T%RoomSet.Cardinality'=1 Destroy T%RoomSet QUIT
; get the room object
SET T%Room=T%RoomSet.RetrieveFirstElement
Destroy T%RoomSet
; default the timestamp to NOW
I P%At="" Create P%At=Base$TimeStamp
SET $RETURN=T%Room.GetReservationOn(P%At)
QUIT

```

The following example demonstrates a request made to a Hospital object to generate a list of all rooms that need cleaning.

```

; Purpose: Produces a cleaning list. A collection of all rooms in need of
cleaning.
;The collection is passed in as input - if no collection passed, one is
created.
; Returns: The collection object containing the rooms in need of cleaning
;
Input: (
  CleaningList:P%List
)
IF $(P%List)=" " Create P%List=Base$List
; Loop through the floors and inquire of each for any rooms that need
cleaning
SET T%FloorObj=" "
FOR SET T%FloorObj=$ORDER(I%FloorList(T%FloorObj)) QUIT:T%FloorObj=" "
DO
  .DO T%FloorObj.GetCleaningList(P%List)
SET $RETURN=P%List
QUIT

```

Events and Messages

When an object receives a message or is notified of an event, one of the parameters can be an object. This object can then be used by the service. Note that the ownership of the object varies depending on the semantics of the service. All events pass the object, which creates the event as a parameter (without transferring ownership).

In the following example, the method is used to enter a charge into an account.

```

;Charge for class account
Input: (P%Service,P%Room,P%Amount)
;P%Service is a Service Object
;P%Room is room for which the service was requested.
;
;Returns the Charge (Ownership retained by account)
;Validate the input
.;Validation code not shown
;Create a charge object
CREATE T%Chg=Charge(P%Service,P%Room,P%Amount)
;Add the charge object to the account
DO I%Log.InsertElement(T%Chg)
;Increment the account balance
SET I%Balance=I%Balance+P%Amount
QUIT

```

Domain Names

All objects are created within some domain and within that domain they can be given unique names. For objects named within a domain, it is possible to refer to that object simply by using its name. The format of a domain name is as follows:

O%[Domain::]Name

Domain variables are persistent, they survive a processes lifetime.

In the following example, all hotels with the domain Ritz are named by the city where they are located.

```
;The following fragment checks the
;current occupancy for the St. Elizabeth hospital in Boston
;
SET T%Occ=O%Elizabeth::Boston.Occupancy
;
```

Defining Objects

The Types of Classes

There are four types of classes:

- **Abstract**
- **Concrete**
- **Nested**
- **Mix-in**

Abstract classes define behavior, but do not create objects. Abstract classes are found in the higher levels of the class hierarchy and as such, act as placeholders, Methods, Properties, Events and Variables can be defined at these levels and inherited by lower levels.

Concrete classes define behavior and are used to create instances. Concrete classes are actually used in the application. They hold Method, Property, Event and Variable definitions as well as inherit them from superclasses. Concrete classes are generally “leaf” classes in the class hierarchy.

Nested classes are classes the reside in their parents name space. Nested classes do not inherit services from the parent. They define a whole new definitional hierarchy within the parent class. They can inherit from other nested classes. They are used to define objects that are private to the parent class.

Mix-in classes are abstract classes that are linked into some point of the class hierarchy where the set of services they contain can be inherited by all of its descendants through

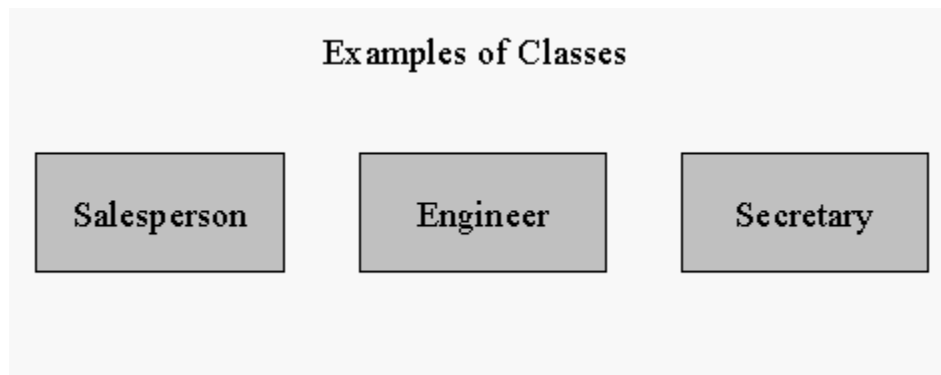
multiple inheritance. They can hold all definitional components of an abstract class. For example, the mix-in class Base\$AbsLockableObject can be linked in at any level of a class hierarchy. At that point, its descendants inherit all database locking services offers.

Inheritance

Inheritance is a mechanism by which a class definition can inherit attributes (variables, methods, properties, or events) from another class.

Inheritance allows you to construct class definitions that include only the specific code needed for that class. Common attributes can be placed in superclasses and specific items are placed in the subclasses.

The end result and major benefit is reusable code and not necessarily from the current project, but previous and parallel projects. Additionally, if the classes have been implemented before in other applications there is a good chance that errors would have been worked out already.



The following partial class definition may be used to define a salesperson object in a sales office application:

Class: Salesperson

Instance vars: Name, DOB, Sex, Address, City, State, Zip, Region, EmployeeNo., SalesTotal, Phone, Ext, Salary

Methods: Modify, GetDemographics, CalcBonus, MakeSale

Should we decide to add a second class, a Secretary class, the definition may look like something below:

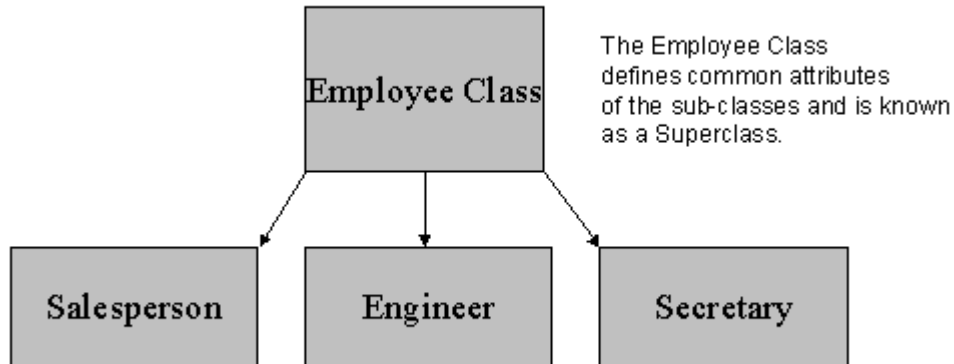
Class: Secretary

Instance vars: Name, DOB, Sex, Address, City, State, Zip, EmployeeNo., Phone, Salary, SkillSet

Methods: Modify, GetDemographics, CalcBonus, AddSkill, HasSkill

Note that both classes, though defining very different types of objects, have much duplication. In fact, lets assume that the GetDemographics method may be the exact same code logic on both classes. Do we have to enter in both classes separately and maintain duplicate code?

Inheritance makes it possible to avoid the duplication.



If we create a third class called Employee and “link” that class to our existing classes, so that the Employee class is a “parent” of the Secretary and Salesperson classes.

This linkage occurs in EsiObjects by a number of methods. These relationships form a tree structure referred to as a *Class Hierarchy* or an *Inheritance Tree*.

A parent class is called a *Superclass* and a child class is referred to as a *Subclass*.

Subclasses automatically inherit all attributes from their parent class(es). Thus we can move the common methods, and variables from the two child classes into the superclass. Objects created from these child classes implement all attributes defined in their own class, and all classes above them in the class hierarchy.

Now there is only one location where the common elements (such as the method GetDemographics) are stored. We only have to maintain it in one area. As other common elements are added, they can be added to the Employee class and are automatically inherited by the subclasses. Additionally, existing objects that were created from the subclasses implement the newly added attributes. Elements that are specific to the sales persons or secretaries can be added to those respective classes.

Multiple Inheritance

EsiObjects supports *multiple inheritance*. This is where a class that has more than one superclass can inherit attributes from all those associated. (Some OO systems do not allow more than one superclass.)

An example of using multiple inheritance would be to take our Salesperson class, which has a superclass of Employee. Let's say we also have a Manager class as seen below:

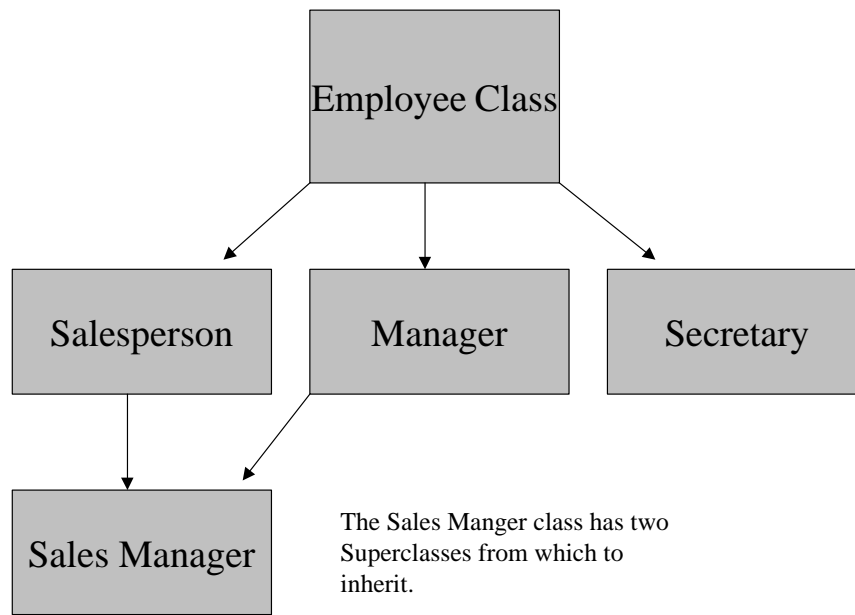
Class: Manager

Superclass: Employee

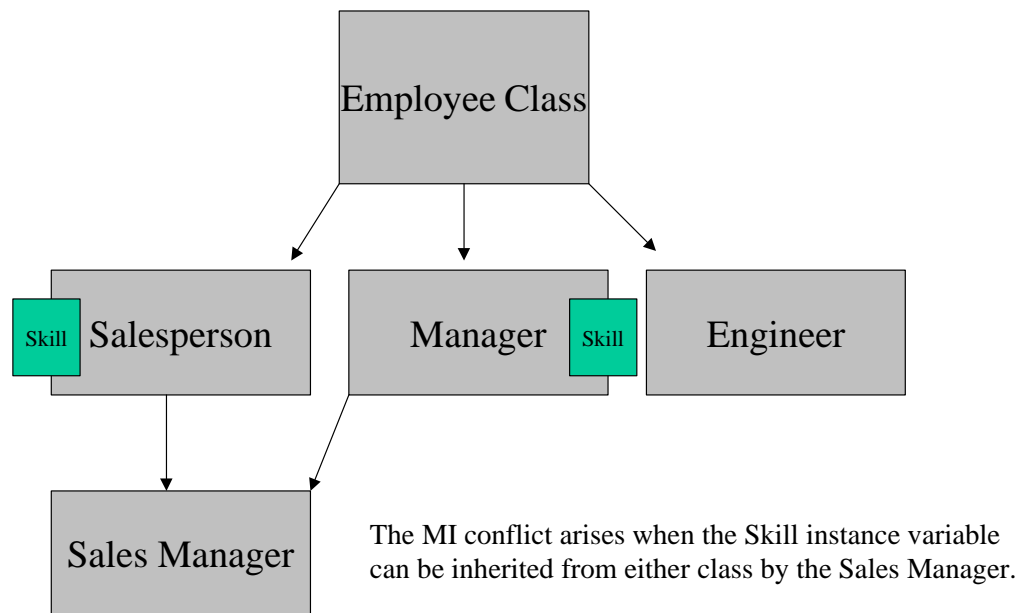
Instance vars: EmployeeList,

Methods: ModifyEmployeeList,

If we decide that we need to add a new employee class, called SalesManager, which is a combination of a salesperson and a manager, we could use multiple inheritance to facilitate this construct.



Note that SalesManager inherits attributes from both Salesperson and Manager. Thus, objects of this class implement both definitions. This is a very powerful capability that allows for greater flexibility in building class definitions. However, this does not come without some potential problems. The most obvious being when a class inherits the same item from both parents. This is known as a *multiple inheritance conflict*.



In EsiObjects, multiple inheritance conflicts are seen as “Error Items”. This is discussed further in the Tools Guide under Session Browser.

Overriding

In the example class hierarchy we showed above, let us assume that the standard employee bonus calculation (stored in the method CalcBonus in the employee class) is fine for secretaries, but may be different for sales persons. Since Salesperson inherits CalcBonus, how do we modify the method for salesperson without affecting the implementation of CalcBonus for the other subclass(es)?

We can *override* the method in the Salesperson subclass and implement the code as needed in that class. Now Salesperson objects that receive the CalcBonus message will run the method defined in the Salesperson class. Secretary objects will invoke the method in the Employee class. Overriding is accomplished by simply creating the item of the same name in the subclass.

Message Searching

As you can see from the discussion above about overriding, when an object receives a message to invoke a particular method or property, a different method may be run depending on what type of object received the message. A Salesperson object that receives the CalcBonus message will run the method in the Salesperson class. A

Secretary object receiving the same message will run the CalcBonus method defined in the Employee class.

An object that receives a message will search for the method or property by following the *inheritance path*. This is the path formed by the path of classes in the inheritance tree, starting from the class that the object belongs to, up to the superclass, and any superclasses beyond.

For example, when the CalcBonus message is sent to an object of Secretary, the object searches first in the class from which it was created. If it cannot find the item defined there, it follows the inheritance path, up to its superclass Employee. It finds the method there and executes the code. If the method were not found in that class, the object would continue searching up the tree to the superclass. Since there is no superclass of Employee, the system would generate an error because the method could not be found.

To avoid generating an error if the method is not found, you can implement a method with the reserved name **\$Unknown**. It must be spelled exactly this way. If the specified method is not found and **\$Unknown** exists, it will be executed in place of the specified method.

Building the Class Hierarchy

When deciding when to add super or subclasses to your hierarchy, there are some guidelines you can follow to determine when and how to build the tree.

A class should be made a subclass of another class only if that class can be considered a “kind of” the superclass. For example, a Salesperson is a “kind of” employee so it would qualify as a proper superclass. If you had a class that defined a purchase order, called PurchaseOrder, it would not be appropriate to make that a subclass of Employee - a purchase order is not a “kind of” Employee. Basically, when linking classes together in super and subclass relationships, ask yourself “Is the sub class a ‘kind of’ the superclass?” If it is, then the relationship is proper. Some cases are not as clear-cut as the example here. In those cases, you may want to consult other OO specialists or use your experiences as a guide. If the relationship is wrong, it will most likely show up in your coding soon enough!

Generic classes are defined at the top of the tree - with more specific class definitions toward the bottom. The specific class is always a subclass of the more general parent class.

If you have class definitions that are very similar and they belong to a similar category (such as Salesperson and Secretary belong to a general category called Employee) that is an indication that you may want to add a superclass to these classes and move the common elements into it. The EsiObject *Promote* option allows you to move attributes up to a superclass. (Point them to this option.)

Do not create super or subclasses for the sake of it. This is a design decision that must be made based on solid criteria that indicates that a superclass is needed. Generally a

superclass is called for when you have 3 classes that could benefit from combining common attributes. 2 classes is a toss up.

The maintenance of the class hierarchy is a critical function and one that can become cumbersome as the tree grows. Management of this hierarchy should be centralized so that decisions to add classes are done consistently and within standard guidelines. Managing the tree to know what's there is critical.

If a method or property is inherited by more than 1 superclass, the system cannot resolve which one should be the inherited item. Thus, the item is marked as an error item and is not implemented by objects of the class.

Avoiding Multiple Inheritance Conflicts

A preferred way to avoid multiple inheritance conflicts is through the use of mix-in classes.

For example, suppose you wanted to add Export/Import capabilities to a variety of different classes, allowing objects of those classes to be saved to, or restored from, a formatted text file. One simple way of adding such capabilities would be to create a mix-in class called **FileExportable**, isolating reusable methods and properties in a common interface called **FileExport**. All the methods and properties in this interface would be specifically related to file import/export operations, and you would adhere to the convention that no class should store unrelated methods or properties in a **FileExport** interface.

This approach would confer two major advantages:

1. Because all methods and properties provided by the FileExportable class would be isolated together in a common FileExport interface, the danger of multiple inheritance conflicts would be greatly reduced.
2. In order to make any class of objects become file exportable, all you have to do is link to FileExportable as a superclass, and possibly implement or override one or two methods/properties pertaining to the specific details of that class. All the rest could be reused from FileExportable.

Resolving Multiple Inheritance Conflicts

One approach to resolving multiple inheritance conflicts in EsiObjects requires the user to *override* the conflicted item. Now the desired code can be defined for the overridden item. Users can call to the superclass implementation by using the **\$SUPER** special variable.

A second resolution is to rename the conflicting item. This approach allows both items to be used by the class. If two items were in question, only one would need to be renamed in order to make them both unique.

Part 3: Reuseability

Using Collection Classes

What Are Collection Classes?

A collection is a special kind of aggregate object that can contain any number of elements. The elements of a collection can be M strings or OIDs of objects. EsiObjects comes with a number of important collection classes in its Base library, each one having distinct characteristics and behavior:

- Set
- Bag
- Array
- List

The above collections are based on the Object Data Management Group's (ODMG) specification.

EsiObjects also includes some other important collections classes:

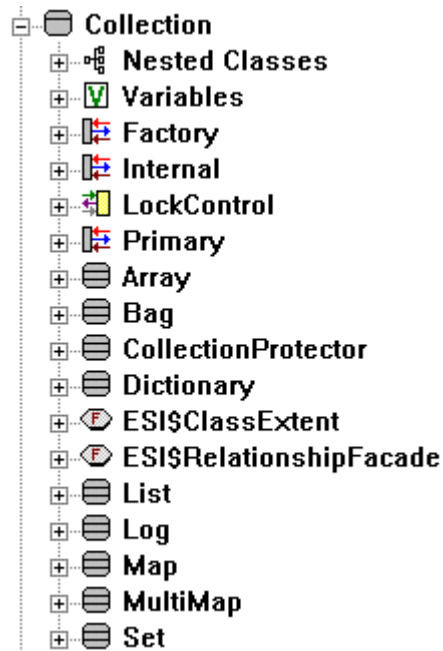
- Map
- MultiMap
- Dictionary
- Log

These collections are not ODMG-specified, although, like all EsiObjects collections, they *are* ODMG-compliant.

Collections Protocol

Collections Hierarchy

This section describes the protocol of each class in the **Collection** hierarchy. The class hierarchy of the collection classes is shown in the following figure.



The root class of all collection classes is the Collection class. The class hierarchy is located in the Base Library. The Collection classes are:

- Array
- Bag
- Dictionary
- List
- Log
- Map
- MultiMap
- Set

The CollectionProtector class is used to protect a collection from inadvertant destruction.

The two classes ESI\$ClassExtent and ESI\$RelationshipFacade are in the ESI library and they inherit the Collection interface. They are not a part of the Collection set of classes.

The following sections summarize the properties and methods for each collection class.

Collection Class

The Collection class is an abstract class. No instances of this class can be created. The class is simply a placeholder for data and methods that can be implemented or shared by its subclasses.

The following is a summary of the properties and methods that are part of the Collection protocol. Many of these methods have no functionality and are intended to be overridden by the subclasses.

Properties

The following is a list of the properties for a Collection class:

Cardinality	Contains the number of elements in the collection.
IsEmpty	Returns 1 (true) if the collection contains no elements, 0 (false) otherwise.
IsOrdered	Returns 1 (true) if the collection is an ordered collection.
AllowsDuplicates	Returns 1 (true) if the collection allows duplicate objects to be inserted in the collection.
AutoDestroy	By default <code>AutoDestroy</code> is <code>False</code> . If <code>AutoDestroy</code> is <code>True</code> then when the <code>RemoveAll</code> method is called or the Collection is destroyed each data item in the collection will in turn be destroyed.

Note: The collection does not add and remove references in any other case.

Example :

```
Create T%Set=Base$Set::AutoDestroy=1
For T%A=1:1:100 Do
. Create T%Tmp=MyLib$MyClass
. Do T%Set.InsertElement(T%Tmp)
. Destroy T%Set
```

All 101 objects are destroyed.

Methods

The following is a list of the methods for a Collection class:

ContainsElement	Returns 1 (true) if the specified element exists in the collection, 0 (false) otherwise.
Copy	Replaces the elements in the specified collection with those of the collection on which the method is performed.
CreateIterator	Returns an ID for an iterator object that can be used to traverse the collection.
InsertElement	Adds the specified element to the collection.
RemoveElement	Removes the specified element from the collection.

RemoveElementAt Removes the element at the location of the specified iterator from the collection.

ReplaceElementAt Replaces the element at the location of the specified iterator with the specified element.

RetrieveElementAt Retrieves the element at the location of the specified iterator.

Set Class

Sets are unordered collections (similar to the Bag collection) that do not allow duplicates.

Methods

The following is a list of the methods for the Set class:

InsertElement Overrides the inherited method from the collection. The specified element is added to the set. If the element is already a member of the set, the method does nothing. No exception is raised.

Union Returns a new Set object whose elements are the union of the elements in the specified set and the set on which the method is performed.

The following example creates two Set objects. One set contains all odd numbers between 1 and 10 and the other set contains all even numbers between 1 and 10.

```
CREATE T%Odds=Base$Set(1,3,5,7,9)
```

```
CREATE T%Evens=Base$Set(2,4,6,8,10)
```

The following example invokes the Union method:

```
SET A$AllNums=AOdds.Union(T%Evens)
```

Intersection Returns a new Set object whose elements are the intersection of the elements in the specified set and the set on which the method is performed.

The following example creates two Set objects. One set contains a list of rooms in a hotel that allow smoking. The other set contains a list of all rooms on the second floor.

```
CREATE T%SmokeRooms(101,105,107,201,208,210,305,401,402)
```

```
CREATE T%Floor2Rooms(201,202,203,204,205,206,207,208,209,210)
```

The following example invokes the Intersection method:

```
SET T%SmokeFree2=T%SmokeRooms.Intersection(T%Floor2Rooms)
```

As a result, T%SmokeFree2 is a Set object that contains elements 210, 208, and 210 (the rooms on the second floor of the hotel that allow smoking).

Difference Returns a new Set object whose elements are the set theoretic difference of the elements in the specified set and the set on which the method is performed.

The following example creates two Set objects. One set contains all prime numbers between 1 and 20. The other set contains all odd numbers between 1 and 20.

```
CREATE T%PRIMES=Base$Set(1,2,3,5,7,11,13,17,19)
```

```
CREATE T%Odds=Base$Set(1,3,5,7,9,11,13,15,17,19)
```

The following example invokes the Difference method:

```
SET T%DiffSet=T%Primes.Difference(T%Odds)
```

As a result, T%Diffset is a Set object that contains the elements 2, 9, and 15 (the difference between the two sets).

Bag Class

Bags are unordered collections (similar to the Set collection) that allow duplicates.

Methods

The following is a list of the methods for a Bag class:

InsertElement Overrides the inherited method from the collection. The specified element is added to the bag. If the element is already a member of the bag, it is inserted a second time.

RemoveElement Overrides the inherited method from the collection. The specified element is removed from the bag. If there is more than one of the specified elements in the Bag, only one of the elements is removed. Because a Bag is an unordered collection, the actual element removed from the physical structure is not specified. If the element to be removed does not exist, the method does nothing. It does not raise an expectation.

Union Returns a new Bag object whose elements are the union of the elements in the specified bag and the bag on which the method is performed.

The following example creates two Bag objects. One contains a student's list of grades from semester one and the other contains a list of grades from semester two.


```
CREATE T%Semester1=Base$Bag(85,85,90,77,93,85)
```

```
CREATE T%Semester2=Base$Bag(91,76,90,90,85)
```

The following example invokes the Union method:

```
SET T%AllGrades=T%Semester1.Union(T%Semester2)
```

As a result, T%AllGrades is a Bag object that contains all 11 grades from both Bag objects.

Intersection

Returns a new Bag object whose elements are the intersection of the elements in the specified bag and the bag on which the method is performed.

The following example creates two Bag objects. One that contains a student's list of grades from semester one and the other contains a list of grades from semester two.

```
CREATE T%Semester1=Base$Bag(85,85,90,77,93,85)
```

```
CREATE T%Semester2=Base$Bag(91,76,90,90,85)
```

The following example invokes the Intersection method:

```
SET T%InterBag=T%Semester1.Intersection(T%Semester2)
```

As a result, T%InterBag is a Bag object that contains elements 90 and 85.

Difference

Returns a new Bag object whose elements are the set theoretic difference of the elements in the specified bag and the bag on which the method is performed.

The following example creates two bag objects. One contains a student's list of grades from semester one and the other contains a list of grades from semester two.

```
CREATE T%Semester1=Base$Bag(85,85,90,77,93,85)
```

```
CREATE T%Semester2=Base$Bag(91,76,90,90,85)
```

The following example invokes the Difference method:

```
SET T%DiffBag=T%Semester1.Difference(T%Semester2)
```

As a result, T%DiffBag is a Bag object that contains elements 91, 76, 90, 85, 77, 93, and 85 (the difference between the two Bag objects).

Array Class

An Array class is a one dimensional dynamically sized grouping of elements. An array is accessible directly through a 1-based integer position specifier.

Methods

The following is a list of the methods for an Array class:

- InsertElementAt** Overrides the inherited method from a collection. The specified element is added to the array at the specified position. The length of the array is resized to fit the new element. All elements from the insertion position and beyond are shifted over a cell. Because of this, insertion into an array can be slow and its use is discouraged.
- RemoveElementAt** Replaces any current value contained in the cell of the array at the specified position with null. It does not remove the cell.
- ReplaceElementAt** Replaces the element stored at the specified position with the specified element.
- RetrieveElementAt** Retrieves the element stored at the specified position in the array. If no element is stored at that cell, null is returned.
- Resize** Sets the size of the array to the specified length. If the size is made smaller, any elements stored in cells beyond the new length are removed.

List Class

A List is an ordered collection that allows duplicates and is implemented as a doubly-linked list. The elements of a list are ordered by the order of their insertion. A list has a head and a tail. It is fast to add or remove an element from the head or the tail of the list, or insert or delete an element from the middle of a list.

Properties

CurrentPosition is a property of the List class. Elements of a list are numbered 1 to n. This property is set as a side effect of each Insert, Remove, Replace, or Retrieve method executed on a list. An Insert method sets CurrentPosition to the position after the newly inserted element. A Remove method sets CurrentPosition to the number of the element preceding the element removed, unless the element removed was the first, in which case CurrentPosition is set to 1. Replace or Retrieve methods set CurrentPosition to the number of the element replaced or retrieved.

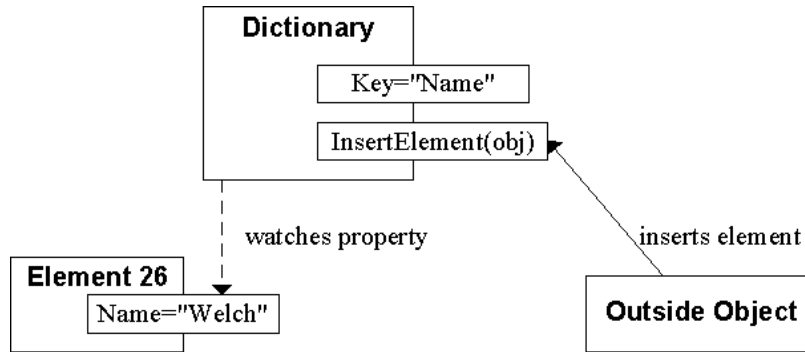
Methods

The following is a list of the methods for a List class:

- InsertElement** Overrides the inherited method from the collection. The specified element is added to the list at `CurrentPosition`. If `CurrentPosition` is not set, the element is added to the end of the list.
- InsertElementAfter** Inserts the specified element after the specified position.
- InsertElementBefore** Inserts the specified element before the specified position.
- InsertFirstElement** Inserts the specified element at the head of the list.
- InsertLastElement** Inserts the specified element at the end of the list. The element that was previously at the end of the list points to this new element.
- RemoveElementAt** Removes the element at the specified position in the list.
- RemoveFirstElement** Removes the element in the first position in the list.
- RemoveLastElement** Removes the element at the end of the list.
- ReplaceElementAt** The specified element replaces the element at the specified position in the list.
- RetrieveElementAt** Returns the element stored at the specified location.
- RetrieveFirstElement** Returns the element stored in the first position in the list.
- RetrieveLastElement** Returns the element stored in the end position in the list.

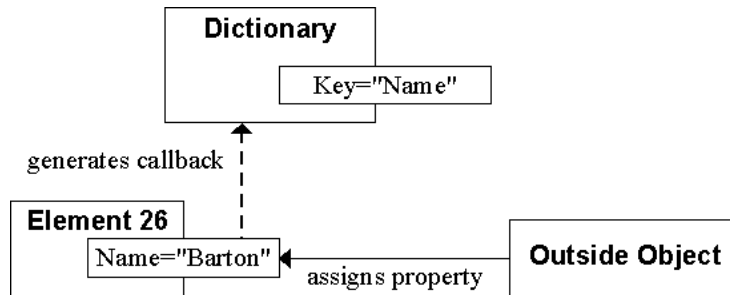
Dictionary Class

A Dictionary is an ordered collection in which the elements are arranged according to the values of a certain common property. This property is known as the "key" property, and all the objects inserted into the dictionary must report a value for this property. Dictionaries watch their elements, so if a property assignment occurs, then the dictionary will automatically update itself to reflect the new property value. This process is illustrated in the following simple diagrams.



The Dictionary's elements are arranged according to the values of a certain property, shared by all the objects. The Dictionary's Key property exposes the property name to external objects.

Whenever a new element is inserted, the Dictionary stores it based on the current value of this property, and issues a watch on the property in case it changes later.



If any object subsequently assigns the value of this property, then a callback is generated to the Dictionary, informing it of the change. It then inquires back to the property, and updates its internal structure accordingly.

Methods

The following is a list of methods implemented by Dictionary:

ContainsElement	True if the Dictionary contains a specific OID.
ContainsKey	True if the Dictionary contains an object whose Key property is equal to the specified value.
Copy	Copies the elements contained within the current Dictionary into a second Dictionary.
CreateIterator	Returns a DictionaryIterator, tied to the current Dictionary.
InsertElement	Inserts a new OID into the Dictionary. The object can only be inserted once, and must implement the Key property. The property value can be null, and multiple elements can have the same property value.

RemoveAll	Removes all the elements in the Dictionary, setting its Cardinality to 0.
RemoveElement	Removes the specified OID from the dictionary, if found.
RemoveElementAt	Accepts a DictionaryIterator as a parameter. Removes the element found at the position specified by the Iterator.
ReplaceElementAt	Accepts a DictionaryIterator as a parameter. Replaces the element found at the specified position with the specified replacement value. The replaced element is removed from the Dictionary.
RetrieveElementAt	Accepts a DictionaryIterator as a parameter. Returns the element at the specified position.
RetreiveElementsByKey	Returns a list of elements whose Key property value equals the specified key value.
RetrieveElementsByPattern	Returns a list of elements whose Key property value matches the specified pattern.

Properties

The following is a list of properties implemented by Dictionary:

AllowsDuplicates	True, since Dictionaries allow the same property value to occur more than once.
Cardinality	Returns the number of elements in the Dictionary
IsEmpty	True if the Dictionary's Cardinality equals 0.
IsOrdered	True, since Dictionaries are ordered by the value of a common property, which must be shared by all elements.
Key	Contains the name of the property used to order the elements contained within the Dictionary.

Log Class

The Log class is a collection in which the elements are arranged according to date/time values associated with each one.

Methods

The following table lists the method implemented by Log.

ContainsElement	Returns true if the Log contains the specified element.
Copy	Copies the elements contained within the current Log into a second Log.
Create Iterator	Returns a LogIterator associated with the current Log.
InsertElement	Inserts a new element into the Log at the current date and time.
InsertElementAt	Inserts a new element into the Log at the specified date and time.
RemoveAll	Removes all elements in the Log.
RemoveElement	Removes the specified OID from the Log.
RemoveElementAt	Accepts a LogIterator as a parameter. Removes the element found at the specified position.
ReplaceElementAt	Accepts a LogIterator as a parameter. Replaces the element found at the specified position with the specified replacement element.
RetrieveElementAt	Accepts a LogIterator as a parameter. Returns the element found at the specified position.
RetrieveTimeStamp	Accepts an OID of an object as a parameter. Returns the time stamp associated with the specified object.

Properties

The following table lists the properties supported by Log.

AllowsDuplicates	Returns true, because Logs allow more than one entry to have the same associated time value. Adding the same object into the Log has no effect.
Cardinality	Returns the number of elements in the Log.
IsEmpty	Returns true if the Cardinality of the Log equals 0.
IsOrdered	Returns true, since Logs are ordered by date/time values.

Map Class

Overview

A Map collection manages a set of ordered pairs, in which each key is associated with exactly one value. A Map permits you to easily find the value associated with a particular key. This collection is similar to MultiMap, which permits more than one value per key.

Creation:

```
CREATE Variable=Base$MultiMap(Argument)
```

AllowsDuplicates property

This property returns true, since Maps allow the same value to be inserted more than once. (Maps only permit one value per key, but the same value can appear under numerous different keys.)

Example

```
IF I%Collection.AllowsDuplicates DO I%Collection.InsertElement(T%Key,T%Value)
```

ContainsElement method

Accepts a value as input, and returns true if that value appears in the Map. This method may require an exhaustive search of all elements in the Map.

Example

```
IF 'I%Map.ContainsElement(T%User) DO I%Map.InsertElement(T%Name,T%User)
```

ContainsKey method

Accepts a key as input, and returns true if that key appears in the Map. This efficient method uses a simple lookup of the Key.

Example

```
IF 'I%Map.ContainsKey(T%Name) DO I%Map.InsertElement(T%Name,T%User)
```

Copy method

Copies the current Map collection into a new target Map. The specified collection must be a Map. It is illegal to use the Copy method to copy the Map into another, heterogeneous collection.

Example

```
DO I%OldMap.Copy(T%NewMap)
```

CreateIterator method

Returns a new MapIterator object.

Example

```
SET T%Iter=T%MyMap.CreateIterator
```

InsertElement method

Inserts the specified key/value combination into the Map. The Cardinality is increased by one.

Example

```
DO I%MyMap.InsertElement(T%Key,T%Value)
```

IsOrdered property

True, because MultiMaps are always ordered.

RemoveAll method

Removes all elements from the Map, resetting its Cardinality to zero. All collection iterators remain stable.

Example

```
DO I%Multi.RemoveAll
```

RemoveElement method

Removes any element from the Map collection matching the specified key. May affect the Cardinality of the collection, and possibly IsEmpty. Any iterators pointing to the specified position will remain stable.

Example

```
DO T%MyMap.RemoveElement(T%Key)
```

RemoveElementAt method

Removes any element from the Map collection matching the specified key or iterator. May affect the Cardinality of the collection, and possibly IsEmpty. Any iterators pointing to the specified position will remain stable.

Examples:

```
DO I%MyMap.RemoveElementAt(T%Key)
```

```
DP I%MyMap.RemoveElementAt(T%Iterator)
```

ReplaceElementAt method

Replaces the item described by the specified iterator with the new specified value. Requires two parameters, an iterator describing the position to be replaced, and the new value for that position.

Example

```
DO T%MyMap.ReplaceElementAt(T%Iterator,T%NewValue)
```

RetrieveElement method

Returns any item having the specified key. The item's value is returned.

Example

```
SET T%Value=I%MyMap.RetrieveElement(T%Key)
```

RetrieveElementAt method

Returns any item matching the specified key or iterator position. The item's value is always returned.

Examples:

```
SET T%Value=I%Map.RetrieveElementAt(T%Iterator)
SET T%Value=I%Map.RetrieveElementAt(T%Key)
```

RetrieveElementsByKey method

Returns a list of items matching the specified key. If no key is specified, then the Map's null position is used. The item's value is always returned.

Examples:

```
SET T%List=T%MyMap.RetrieveElementsByKey(T%Key)
SET T%List=T%MyMap.RetrieveElementsByKey
```

RetrieveElementsByPattern method

Returns a list containing all elements whose values match the specified pattern.

Example

```
SET T%List=I%Map.RetrieveElementsByPattern("la.29anp")
```

RetrieveKeysByPattern method

Returns a list containing all elements whose keys match the specified pattern.

Example

```
SET T%List=I%Map.RetrieveKeysByPattern("la.29anp")
```

MultiMap Class

Overview

A MultiMap collection manages a set of ordered pairs, in which each key is associated with one or more values. A MultiMap permits you to easily find all of the values associated with a particular key. This collection is similar to Map, which only permits one value per key.

The keys in the collection are ordered according to the normal sorting sequence. If multiple values contain the same keys, then no particular ordering among them is guaranteed.

Creation

```
CREATE Variable=Base$MultiMap
```

AllowsDuplicates property

This property returns true, since MultiMaps allow the same value to be inserted more than once.

Example

```
IF I%Collection.AllowsDuplicates DO I%Collection.InsertElement(T%Key,T%Value)
```

ContainsElement method

Accepts a value as input, and returns true if that value appears in the MultiMap. This method may require an exhaustive search of all elements in the MultiMap.

Example

```
IF ' I%MultiMap.ContainsElement(T%User) DO I%MultiMap.InsertElement(T%Name,T%User)
```

ContainsKey method

Accepts a key as input, and returns true if that key appears in the MultiMap. This efficient method uses a simple lookup of the Key.

Example

```
IF ' I%MultiMap.ContainsKey(T%Name) DO I%MultiMap.InsertElement(T%Name,T%User)
```

Copy method

Copies the current MultiMap collection into a new target MultiMap. The specified collection must be a MultiMap. It is illegal to use the Copy method to copy the MultiMap into another, heterogeneous collection.

Example

```
DO I%OldMulti.Copy(T%NewMulti)
```

CreateIterator method

Returns a new MultiMapIterator object.

Example

```
SET T%Iter=T%MyMultiMap.CreateIterator
```

InsertElement method

Inserts the specified key/value combination into the MultiMap. The Cardinality is increased by one.

Example

```
DO I%MyMultiMap.InsertElement(T%Key,T%Value)
```

IsOrdered property

True, because MultiMaps are always ordered.

RemoveAll method

Removes all elements from the MultiMap, resetting its Cardinality to zero.

Example

```
DO I%MyMultiMap.RemoveAll
```

RemoveElement method

Removes all elements from the MultiMap collection matching the specified key. May affect the Cardinality of the collection, and possibly IsEmpty. If any iterators point to the affected position, then they will be reset.

Example

```
DO T%MyMultiMap.RemoveElement(T%Key)
```

RemoveElementAt method

Removes all elements from the MultiMap collection matching the specified key or iterator. (A key will remove all references to the given key while an iterator will remove only the item that the iterator currently points to.) May affect the Cardinality of the collection, and possibly IsEmpty. If any iterators point to the affected position, then they will be reset.

Examples:

```
DO I%MyMulti.RemoveElementAt(T%Key)
```

```
DO I%MyMulti.RemoveElementAt(T%Iterator)
```

ReplaceElementAt method

Replaces the item described by the specified iterator with the new specified value. Requires two parameters, an iterator describing the position to be replaced, and the new value for that position.

Example

```
DO T%MyMultiMap.ReplaceElementAt(T%Iterator, T%NewValue)
```

RetrieveElement method

Returns a list of items having the specified key.

The first position in the list contains the key specified as an argument, and the remaining positions contain any matching item values. The values are not ordered in any special way.

Note that the returned list will always contain at least one element (the specified key). The second element in the list is the first matching value.

Example

```
SET T%List=I%MyMultiMap.RetrieveElement(T%Key)
```

RetrieveElementAt method

Returns a list of items matching the specified key or iterator position.

The first item of the list is always the key of the items in the remaining list positions.

If the argument is a key, then all items having that key are added to the list. If the argument is an iterator, then the single item at that position is added.

Examples:

```
SET T%List=I%MultiMap.RetrieveElementAt(T%Iterator)
SET T%List=I%MultiMap.RetrieveElementAt(T%Key)
```

RetrieveElementsByKey method

Returns a list of the items matching the specified key. If no key is specified, then the MultiMap's null position is used.

The first position in the list contains the specified key, and the remaining positions contain the associated values.

Examples:

```
SET T%ReturnList=T%MyMap.RetrieveElementsByKey(T%Key)
SET T%ReturnList=T%MyMap.RetrieveElementsByKey
```

RetrieveElementsByPattern method

Returns a list containing all elements whose values match the specified pattern.

Example

```
SET T%List=I%MultiMap.RetrieveElementsByPattern("1a.29anp")
```

RetrieveKeysByPattern method

Returns a list containing all elements whose keys match the specified pattern.

Example

```
SET T%List=I%MultiMap.RetrieveKeysByPattern("1a.29anp")
```

Choosing a Collection Class

The collection that you choose for your particular application depends on a number of factors. These factors include the following features of a collection:

- Order
- Indexing
- Performance of certain methods

The following table describes the characteristics of each collection class. Keep these points in mind when deciding which collection class to use.

Class	Order	Indexed	Insertion Speed	Search Speed	Duplicate Values
Set	None	No	Fast	Fast	No
Bag	None	No	Fast	Fast	Yes
Array	Insertion / index	By integer	Slow	Slow	Yes
List	Insertion	No	Fast	Slow	Yes
Map	By key	By key	Fast	Fast	Yes
MultiMap	By key	By key	Fast	Fast	Yes
Dictionary	By property	By property	Fast	Fast	No
Log	By time	By time	Fast	Fast	Yes

In the previous table, the term *ordered* means the order in which elements are traversed or removed in the collection, if any. This is discussed below. The term *indexed* means that the items in the collection can be retrieved by some associated index value.

Columns 4 and 5 describe the performance of each class. In applications that require many insertions into the collection, insertion speed is important. In other applications, lookup speed may be more important. "Insertion" is taken to mean insertion at *any* position within the collection—not necessarily at the end, where insertion speed improves for some collections.

A collection such as a Bag is "unordered," because its elements cannot be relied upon to come out in any specific order. (Note that an unordered collection is not guaranteed to be randomly ordered, either.) However, you can rely on there being some arbitrary order, and this may be useful to you when traversing the elements of the collection. Any specific assumptions about the order in which elements are sorted is considered invalid. This means that the ordering is completely arbitrary, and subject to change.

Iterators have a contract regarding traversal of the collection, which varies according to whether the iterator is ordered. See the section on iterators for more details.

Creating Collection Objects

A collection, like any object, is instantiated with the **CREATE** command. However, a collection can be created as a component of a parent object. In that case, it is part of the class definition for that object and the collection becomes a part of the encapsulated state of the parent; it is automatically created and destroyed along with its parent. Also, an application can dynamically **CREATE** a collection at runtime to store its own elements or to pass off to an external object.

The following are different ways that you can create a collection:

- Inline code** A method can explicitly contain a **CREATE** command that creates the collection. This is a flexible approach: a handle to the collection can be created in any kind of variable.
- Create Binding** An instance variable can be defined as an automatically-created collection. In the variable definition editor, set the variable's *binding* to **CREATE**, and its class to the appropriate collection class.
- Static Binding** An instance variable can be defined as having a static binding, and created manually in the parent's **CREATE** method. This is particularly useful when special setup work is required, such as inserting a number of elements into the collection when the parent is created.

For more about variable definitions, see the section on the [Variable Definition Editor](#) in the [EsiObjects Tools Guide](#).

All EsiObjects collections can be created with an initial set of elements placed in the collection. These elements are specified as parameters on the **CREATE** command. For example, the following command creates a Set object with six elements:

```
CREATE I%PrimeNumbers=Base$Set(2,3,5,7,11,13)
```

There is no explicit limit on the number of elements that you can specify on the **CREATE** command. The number is restricted only by the maximum length of a command line in the underlying M platform. Note that the length of an EsiObjects line is generally shorter than the length of the resulting M line.

In the following example, a **Bag** collection is created with five elements:

```
CREATE I%Grades=Base$Bag(85,93,79,85,90)
```

The following command creates an array object and places **4** elements in the array:

```
CREATE I%Heading=Base$Array("Date","Time","Page","Title")
```

All the collections support insertion of elements during creation.

Manipulating Collection Objects

Collection Life Cycle

Like any other object, collection objects have a life cycle of creation, lifetime, and destruction. Collections can be destroyed explicitly with the **DESTROY** command. If the collection is a sub-component of another object, then the collection is removed when the object containing it is destroyed.

Destroying a collection has no effect on any objects that are elements of the collection. To destroy its elements, see the section on [Deleting all Objects in a Collection](#), below.

Accessing all Elements in a Collection

The elements in a collection can be accessed manually, or by using an *iterator*. This section describes the manual process.

List collections use a position indicator to point to a given position. **Array** collections use an index. To manually access all elements in these collections, simply retrieve them in sequential order at each of the positions in the collection.

The following lines can be placed inside a method, or executed in sequence from the EsiObjects Xecute Shell or an object browser. They create an Array object, populating it with values, and then access all members of the array, placing them in the output window. (Make certain the output window is visible first.)

```
CREATE I%Array=Base$Array( "Monday" , "Tuesday" , "Wednesday" , "Thursday" , "Friday" , "Saturday" , "Sunday" )
FOR T%X=1:1:I%Array.Length DO $ENV.Output(I%Array.RetrieveElementAt(T%X))
DESTROY I%Array
KILL I%Array
```

Note that the Array implements an unusual **Length** property, making it possible to determine how many positions have been allocated. The following example is similar, except it uses a **List**. Note that the **Cardinality** property is used to determine the number of elements. (All collections have a Cardinality property.)

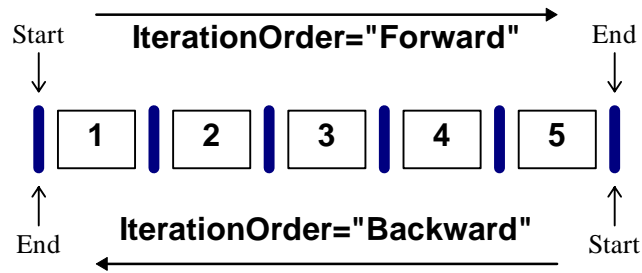
```
CREATE I%List=Base$List( "Uno" , "Dos" , "Trés" , "Cuatro" , "Cinco" , "Seis" , "Siete" , "Ocho" , "Nueve" , "Diéz" )
FOR T%X=1:1:I%List.Cardinality DO $ENV.Output(I%List.RetrieveElementAt(T%X))
DESTROY I%List
KILL I%List
```

Only List and Array permit manual access to all their elements, because only these collections use sequential numbers for lookup purposes. All collections use an object called an iterator to traverse their elements. Generally speaking, the use of iterators is the preferred way to sequentially access collection elements.

Iterators

What is an Iterator?

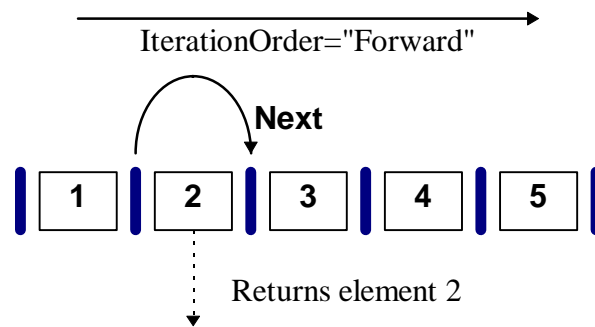
All EsiObjects collections use iterators to sequentially traverse their elements. The iterator is a special object that establishes a unique relationship with the collection, permitting extra-efficient traversal. Thus, the use of iterators is always the preferred way for an external object to loop through the elements in a collection, and for many kinds of collections it is the only way.



In this diagram, the vertical blue stripes denote iterator positions, while the numbered boxes represent elements in the collection.

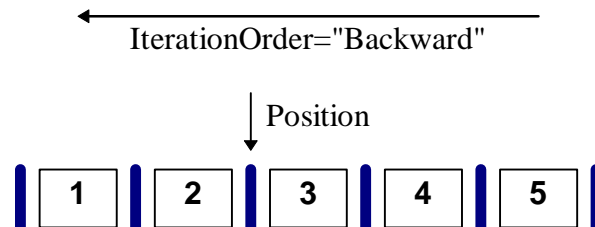
This diagram shows an ordered collection containing five elements. If the iterator's direction is "Forward", it will proceed sequentially from items one through five. If the iterator's position is "Backward", then it will proceed from five to one. The iterator is always considered to be "between" elements, as shown by the thick blue horizontal bars.

The iterator's **Next** method returns the next item in the collection, and moves the iterator ahead one position. This is shown in the following diagram.

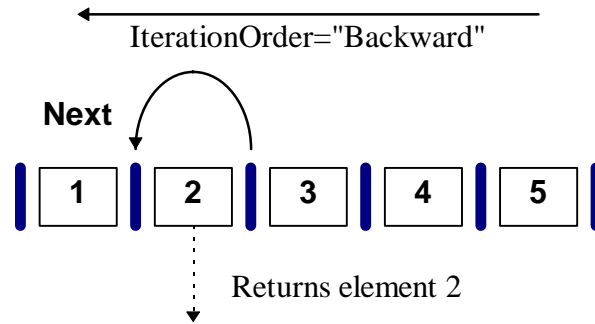


In this diagram, the iterator starts between elements 1 and 2 of an ordered collection such as a **List**. The **Next** method is invoked. The iterator's position is advanced forward one element, leaving it between elements 2 and 3. The **Next** method returns the value of element 2 (without removing it).

Next, the iterator's direction is changed to **Backward**. The following diagram illustrates this situation. The iterator is still in the same position, between elements 2 and 3, but the direction has changed.



Finally, the **Next** method is once more invoked. The iterator moves backward one element, from the point between elements 2 and 3, to the point between elements 1 and 2. The value of element 2 is returned. This is illustrated in the following diagram.



Some Simple Iterator Examples

The following example illustrates iterators for the **Set** class. A set of prime numbers is created in the instance variable **I%Primes**, and a special **SetIterator** object is used to access elements from the collection. Many of the commands use the output window, so make sure that it is visible before trying this example.

The lines of code in this example can be entered into a method, or can be typed sequentially from the Xecute Shell or an Object Browser.

First, a new set is created, and a set iterator is returned.

```
CREATE I%Primes=Base$Set(2,3,5,7,11,13,17,19,23,29,31,37)
SET I%Iterator=I%Primes.CreateIterator
```

Next, the first and last elements of the set are displayed in the output window.

```
DO $ENV.Output("First: "_I%Iterator.First)
DO $ENV.Output("Last: "_I%Iterator.Last)
```

Next, the iteration order is set to forwards, and the iterator is reset to the beginning. (Resetting is necessary because the last element was most recently accessed, above.) All elements are traversed in forwards order. Note that it is not necessary to tell the iterator which element to use—it keeps track of that information automatically.

```
SET I%Iterator.IterationOrder="F" ; Forward (F or 1)
DO $ENV.Output("Order: "_I%Iterator.IterationOrder)
DO I%Iterator.Reset
DO $ENV.Output("-----")
FOR SET T%Item=I%Iterator.Next QUIT:T%Item="" DO $ENV.Output(T%Item)
```

Next, the iteration order is set to backwards, and all elements are traversed in reverse order. (In this case, it is not necessary to reset the iterator, because it is already at the end following the most-recent traversal.)

```
SET I%Iterator.IterationOrder="B" ; Backwards (B or -1)
DO $ENV.Output("-----")
DO $ENV.Output("Order: "_I%Iterator.IterationOrder)
FOR SET T%Item=I%Iterator.Next QUIT:T%Item="" DO $ENV.Output(T%Item)
```

IterationOrder Property

The **IterationOrder** property returns the direction in which the collection will be traversed by the iterator. Valid *assignment* values are as follows:

- "F" or 1 for Forward
- "B" or -1 for Backward

Accessing the *value* of the **IterationOrder** property returns "Forward" or "Backward".

For unordered collections (Bag or Set), the order of iteration is arbitrary, though not random. For ordered collections (List, Array, etc.), the order is fixed, and backward traversal should be the opposite of forward traversal. By default, **IterationOrder** is **Forward**.

Example

The lines of code in this example can be entered into the Xecute shell or an object browser. Alternatively, they can be entered into a scratch method.

This example illustrates the use of the IterationOrder property in a List object. A List containing five elements is created, and an iterator to the list is returned. The iterator is used to traverse the first two elements in the List.

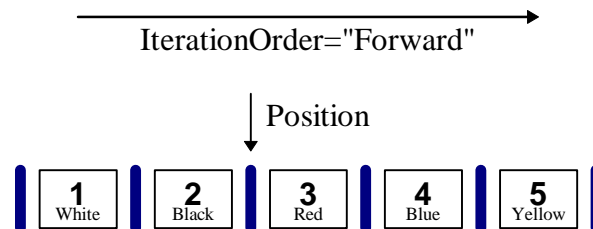
```
CREATE I%List=Base$List("White", "Black", "Red", "Blue", "Yellow")
SET I%Iterator=I%List.CreateIterator
DO $ENV.Output(I%Iterator.Next)
```

The above line returns the first element, White.

```
DO $ENV.Output(I%Iterator.Next)
```

The above line returns the second element, Black.

Here is the current status of the iterator, at this point.



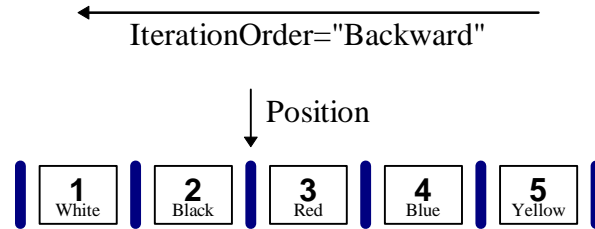
In this diagram, the vertical blue stripes denote iterator positions, while the numbered boxes represent elements in the collection.

The iterator's position is between elements 2 and 3 of the list. Its direction is Forward, meaning that the next element is element 3. Invoking the Next method would return the third element, Red. However, let's instead change the iteration order.

Changing the iteration order affects the direction in which the iterator moves through the collection. The following line of code changes the order to Backward.

```
SET I%Iterator.IterationOrder="B"
```

The following diagram summarizes the status of the iterator, following this change.



Now the iterator is still positioned between elements 2 and 3 in the collection, but the direction is backward. Thus, the next element would be element 2 (Black.) The following line of code returns the next element.

```
DO $ENV.Output(I%Iterator.Next)
```

Note that element 2, Black, is again returned.

Next Method

The Next method returns the value of the next element in the collection, as defined by the position of the iterator. It also advances the iterator one element ahead in the collection.

If there is no next element, this method returns null. (Note: because many collections can contain null elements, testing to see whether a null element is returned may not be a reliable way to determine whether the collection is empty. See the **More** method for further details.)

The direction of iteration depends on the **IterationOrder** property. If **IterationOrder** is **Forward**, then **Next** will always move one position ahead in the collection. If **IterationOrder** is **Backward**, then **Next** will move in reverse.

Example

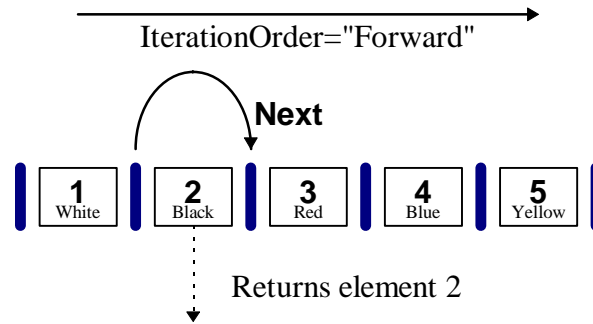
The lines of code in this example can be entered into the Xecute shell or an object browser. Alternatively, they can be entered into a scratch method.

This example illustrates the use of the **Next** method in a **List** collection. A List containing five elements is created, and an iterator to the list is returned. The iterator is used to traverse the first two elements in the List.

```
CREATE I%List=Base$List("White", "Black", "Red", "Blue", "Yellow")
SET I%Iterator=I%List.CreateIterator
DO $ENV.Output(I%Iterator.Next)
```

The above line returns the first element, White.

The following diagram shows what happens next. The iterator is positioned between the first and second list elements. Invoking the **Next** method moves the iterator one position forward, to the point between the second and third elements. The diagram illustrates this.



In this diagram, the vertical blue stripes denote iterator positions, while the numbered boxes represent elements in the collection.

```
DO $ENV.Output(I%Iterator.Next)
```

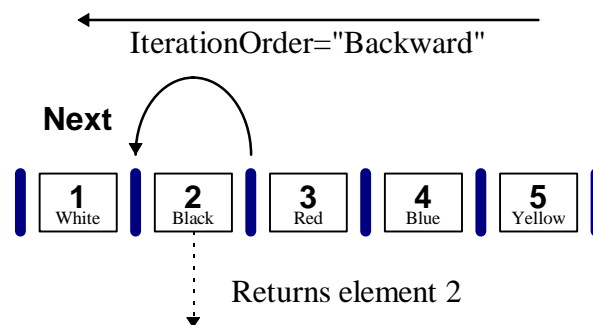
The above line returns the second element, Black, as shown in the diagram.

Now the iterator's position is between elements 2 and 3 of the list. Its direction is Forward, meaning that the next element is element 3. Invoking the Next method would return the third element, Red. However, let's instead change the iteration order.

Changing the iteration order affects the direction in which the iterator moves through the collection. The following line of code changes the order to Backward.

```
SET I%Iterator.IterationOrder="B"
```

Now the iterator is still positioned between elements 2 and 3 in the collection, but the direction is backward. Thus, the next element would be element 2 (Black.) The following line of code returns the next element. The diagram illustrates this.

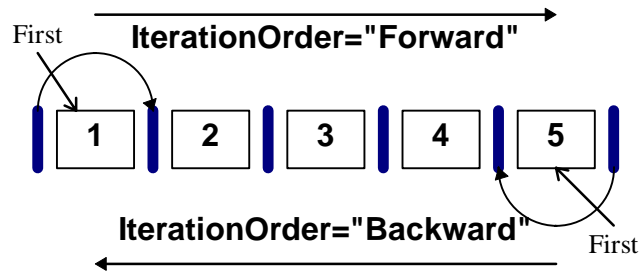


```
DO $ENV.Output(I%Iterator.Next)
```

Note that element 2, Black, is again returned.

First Method

This method returns the *first* element in the collection, based on the current iteration order. The iterator is positioned *after* the returned element.



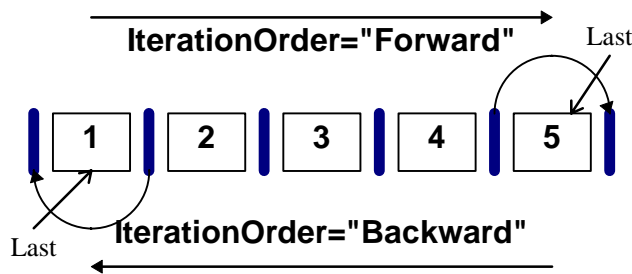
In this diagram, the vertical blue stripes denote iterator positions, while the numbered boxes represent elements in the collection.

- if the iteration order is forward, then the *first* element in the collection is returned
- if the iteration order is backward, then the *last* element in the collection is returned

This method is equivalent to invoking the **Next** method immediately after the **Reset** method.

Last Method

This method returns the *last* element in the collection, depending on the iteration order. The iterator is positioned at the *end* of the collection. The following diagram illustrates this method.



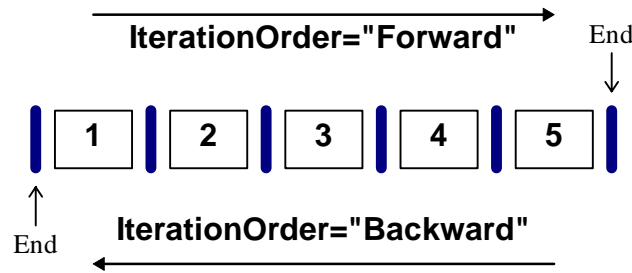
In this diagram, the vertical blue stripes denote iterator positions, while the numbered boxes represent elements in the collection.

The Last method makes it possible to position the iterator at the end of the method. This can be accomplished by invoking the Last method, while ignoring its return value.

```
DO T%Iterator.Last ; Sets position to end.
```

More Method

This method returns *true* if there are *more* elements to be traversed in the collection, as defined by the iterator's position. The following diagram summarizes the ending position of an iterator in an ordered collection. Note that the end position follows the last element of an ordered collection if the iteration order is forward, but precedes the first element if the iteration order is backward.



In this diagram, the vertical blue stripes denote iterator positions, while the numbered boxes represent elements in the collection.

This method returns a true or false value, based on the following conditions:

- if the iterator's position is at the end of the collection, *false* is returned
- otherwise, *true* is returned.

Example

The lines of code in this example can be entered into the Xecute shell, an object browser, or into a scratch method.

First, a Bag collection is created, containing twelve different elements: the names of the months of the year.

```
CREATE I%Months=Base$Bag("January","February","March","April","May","June","July","August","September","October","November","December")
```

Next, an iterator is created for the Bag collection.

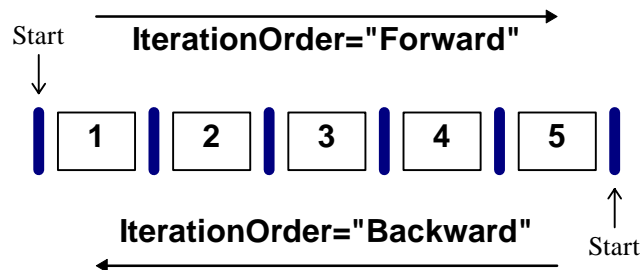
```
SET I%Iter=I%Months.CreateIterator
```

Finally, a FOR loop is used to list the collection's elements to the output window.

```
FOR QUIT:'I%Iter.More DO $ENV.Output(I%Iter.Next)
```

Reset Method

This method **resets** the iterator to the starting position of the collection. Depending on the iteration order, this position can either precede the first element, or follow the last element, of an ordered collection. The following diagram summarizes the iterator's starting position.



In this diagram, the vertical blue stripes denote iterator positions, while the numbered boxes represent elements in the collection.

Immediately after the **Reset** method has been invoked, both the **Next** and **First** methods would return the same value (the first element in the collection, as determined by its current iteration order.)

Using Iterators

Iterating a Collection

Iterators work the same ways for all collections. They can be used to conveniently loop through (or iterate) the elements in the collection.

The code in this example is intended to be used in a scratch method. If you would like to try it out, then just create your own temporary class in a User library (do *not* use one of the system-level libraries, such as ESI or Base) and create a new method of that class. This example will assume that a class called **Test** in the **User** library is created, and that the method **IterationTest** is added to this class.

The method's code is shown below. Note that this code can be copied and pasted directly into a method editor.

```

; Method - User$Test.Primary::IterationTest
Input:() ; This method accepts no parameters.
; First, create a List collection containing the days of the week.
CREATE T%Collection=Base$List("Monday","Tuesday","Wednesday","Thursday","
Friday","Saturday","Sunday")
; Next, obtain an iterator to the collection.
SET T%Iter=T%Collection.CreateIterator
; Display the collection's last item.
DO $ENV.Output("Last item: "_T%Iter.Last)
; Display the collection's first item.
DO $ENV.Output("First item: "_T%Iter.First)
; Now, we'll set the iterator to work backwards.
SET T%Iter.IterationOrder="B"
DO $ENV.Output("Iteration order: "_T%Iter.IterationOrder)
; Reset the iterator, to ensure it's at the beginning position.
DO T%Iter.Reset
; The following loop displays a report to the output window.
; Appropriate information is displayed for all collection elements.
DO $ENV.Output("Days of week report:")
FOR QUIT:'T%Iter.More DO
. ; Get next day...
. SET T%Day=T%Iter.Next
. ; Weekend or weekday?...
. SET T%Type=$select($extract(T%Day)="S":"Weekend",1:"Weekday")
. ; Display relevant information, indented 2 sp...
. DO $ENV.Output("  "_T%Day_" <"_T%Type_">")
DO $ENV.Output("(end of week)")
; Cleanup work follows...
DESTROY T%Iter,T%Collection
QUIT ; End of method.

```

This method creates a collection containing the days of the week. It then creates an iterator to the collection, and puts the iterator through the paces by invoking every one of its behaviors (methods and properties). The results are displayed in the Output window, so it's a good idea to make certain this window is visible before executing the method.

From the Xecute shell, enter the following two commands:

```

CREATE I%Scratch=User$Test
DO I%Scratch.IterationTest

```

The following text should appear in the output window:

Last item: Sunday

First item: Monday

Iteration order: Backward

Days of week report:

Sunday <Weekend>

Saturday <Weekend>

Friday <Weekday>

Thursday <Weekday>

Wednesday <Weekday>

Tuesday <Weekday>

Monday <Weekday>

(end of week)

Iterators generally work the same way for all collections, with two exceptions:

- each collection has different characteristics, and this is reflected in the iteration order
- some collections have special capabilities, and their iterators have been enhanced to provide supporting functionality

Multiple Iterators

One collection can have multiple iterators. This may occur when several different objects are iterating through the same collection in different ways, or when a single object has more than one reason to iterate the collection. Generally speaking, the iterators will always remain stable under these circumstances. For example, multiple iterators do not directly affect each other's position within the collection, and will not cause problems.

Removal of Elements

Sometimes, the removal of an element may affect an existing iterator whose current position is at the removed element. Even in this case, the iterator's behavior is stable. For example, invoking the Next method simply returns the next element after the deleted element.

Two principles apply to the state of the iterator:

- Any behaviors that depend on the iterator's current position will reflect a "snapshot" of the collection, the last time it was accessed;
- Any time the iterator's position is moved, its new position will be based on the collection's current state at the time of iteration.

Destroying the Collection

For example, if a collection is destroyed, the iterator's internal state does not change until the next iteration occurs. At that time, the iterator will notice that the collection has been

destroyed. From then on it will behave as though all collection elements had been exhausted.

Example

The following example illustrates the use of multiple iterators. The first iterator is used to traverse the collection forwards, and the second iterator is used to traverse the collection backwards.

The commands in this example may be entered from the Xecute shell, or an object browser, or they may be entered into a scratch method.

Note: this simple example uses the days of the week, in German. You don't need to understand German in order to understand this example.

```
CREATE I%List=Base$List("Montag", "Dienstag", "Mittwoch", "Donnerstag", "Freitag", "Samstag", "Sonntag")
SET I%IterOne=I%List.CreateIterator
SET I%IterOne.IterationOrder="F"
SET I%IterTwo=I%List.CreateIterator
SET I%IterTwo.IterationOrder="B"
```

Thus far, we have created a list containing the days of the week, in German. There are two iterators. The first will move forward through the list, the second backward. The following FOR loop causes values to be sent to the output window, so make sure it is visible.

```
FOR QUIT:'I%IterOne.More DO $ENV.Output(I%IterOne.Next_/"_I%IterTwo.Next)
```

The output window should reflect the following values:

Montag/Sonntag

Dienstag/Samstag

Mittwoch/Freitag

Donnerstag/Donnerstag

Freitag/Mittwoch

Samstag/Dienstag

Sonntag/Montag

Note that the first iterator produces the days of the week in forward order, while the second produces them in reverse order. Neither iterator affects the other.

Iterating a Set

Because *Sets have no specific order*, iteration order through a Set is arbitrary. It is also subject to be changed in the future. Thus, any programming decisions made about iteration order through sets (except as described below), could possibly be invalidated by future releases of EsiObjects. In fact, during the past few years the internals of the Set

class have been restructured several times to make the collection more robust—the iteration order of Sets has changed as a result, and may continue to do so in the future.

Iteration begins at the start of the Set, and returns every element in the Set as the **Next** method is repeatedly invoked. If elements are removed during the iteration process, then they will not be visited after their removal from the Set. Any elements inserted into the Set are guaranteed to be traversed if iteration continues exhaustively.

Changing Iteration Order

When iterating through a Set, changing the **IterationOrder** property causes the direction of the iteration to change and returns elements in the reverse direction. (However, since the collection is unordered, it is impossible to generalize further). Iteration ends at the point where the iteration order was initially changed.

In this example, the numbers 1 through 10 are inserted into a Set in random order.

```
CREATE I%NumSet=Base$Set(8,2,7,1,10,4,6,3,9,5)
```

An iterator is next created.

```
SET I%Iterator=I%NumSet.CreateIterator
```

Now five more elements are traversed. (Since there is no guarantee about Set order, it is impossible to say which items will be returned—different versions of EsiObjects may produce different behaviors.)

```
FOR T%X=1:1:5 SET T%Value=I%Iterator.Next DO $ENV.Output(T%Value)
```

Next, the iterator's order is changed to backwards, and a dividing line is sent to the output window.

```
SET I%Iterator.IterationOrder="B"
DO $ENV.Output("-----")
```

Now all of the elements in the Set are iterated, in reverse order. Notice that all the elements are now seen in the output window.

```
FOR QUIT:'I%Iterator.More DO $ENV.Output(I%Iterator.Next)
```

If elements are *added* to the set after the iteration process has commenced, then you can safely assume that the iterator will still encounter those elements. The following statements illustrate this.

We'll reset the iterator and output another dividing line, in order to start over.

```
DO I%Iterator.Reset
DO $ENV.Output("-----")
```

Next, we'll iterate through the first eight elements.

```
FOR T%X=1:1:8 DO $ENV.Output(I%Iterator.Next)
```

Now we'll add a new element to the set.

```
DO I%NumSet.InsertElement(1.5)
```

Continuing the iteration process to the end, notice that the newly-added element is also returned.

```
FOR QUIT:'I%Iterator.More DO $ENV.Output(I%Iterator.Next)
```

Iterating a Bag

Because Bags have no order, the order of elements returned in iterating through a Bag is arbitrary. It is also subject to be changed in the future, except as described below.

Think of a Bag collection as a large sack into which items are carelessly tossed. There is no particular order to the items, and no guarantee that two of the items will not look exactly the same. *Think of a bag iterator as a sack-management consultant that you must hire to shuffle through the items in the sack.* You will be shown one item after another, without actually removing the items from the sack. If anyone else happens to add items to the sack during this process, then the consultant is responsible for making sure that you don't miss out on those items.

If you want to make *any* assumptions about their order, even assuming they are randomized, then use a different kind of collection instead. If you insert ten things into a bag and iterate through it, you will get the same things back in some arbitrary order. If you reverse direction, you will get the same things back in a different order. If items are inserted into the bag partway through iteration, then the iterator will definitely return them if you continue to Next through the collection exhaustively. *You can trust the iterator to perform robustly, and to remain continuously stable, no matter what happens to the bag during the iteration process.*

When iterating through a Bag, changing the **IterationOrder** property causes the direction of the iteration to change. Also, iteration continues until it reaches the position where the iteration order was changed. You will not get back the same value twice, unless the same value was inserted into the bag more than once.

The following example creates a **Bag** object containing the numbers **85, 76, 90, 85,** and **100**, and creates an **Iterator** object to traverse the bag:

```
CREATE I%Grades=Base$Bag(85,76,90,85,100)
SET I%Iterator=I%Grades.CreateIterator
Now we'll invoke the Next method twice to see the first two elements in the
bag.
DO $ENV.Output(I%Iterator.Next)
DO $ENV.Output(I%Iterator.Next)
Now, we'll turn the iterator Backwards.
SET I%Iterator.IterationOrder="B"
DO $ENV.Output("-----")
```

Finally, we'll use a **FOR** loop to traverse all the elements in the bag. Note that none of the elements are skipped—the iterator reaches the start of the bag, then starts over from the end and continues until it reaches the point at which the direction was reversed.

```
FOR QUIT:'I%Iterator.More DO $ENV.Output(I%Iterator.Next)
```

If elements are *added* to the bag after the iteration process has commenced, then you can safely assume that the iterator will still encounter those elements.

Iterating an Array

An **Array** is an ordered collection having numeric indexes. Arrays have a fixed number of cells: failure to populate an array cell will result in a null value being returned for that cell. Iteration begins at cell number 1, and continues forward to the last cell in the array. The value of each cell is returned.

In the following example, an Array is first created containing four elements...

```
CREATE I%Array=Base$Array(10,20,30,40)
```

Next, a new 8th element is inserted...

```
DO I%Array.InsertElementAt("Inserted",8)
```

An iterator for the array is created...

```
SET I%ArrayIter=I%Array.CreateIterator
```

Finally, a loop is used to traverse the array. Notice that, since the fifth through seventh cells are empty, they will report null values.

```
FOR QUIT:'I%ArrayIter.More DO $ENV.Output(I%ArrayIter.Next)
```

Iterating a List

Lists are ordered collections. Their indexes are numeric—the first element is numbered 1, the second is numbered 2, and so on. New elements are generally added to the end of the list, but it is always possible to insert elements before this point.

In the following example, a List is first created containing four elements...

```
CREATE I%List=Base$List(10,20,30,40)
```

Next, a new 3rd element is inserted, causing the subsequent elements to be bumped back in the list.

```
DO I%List.InsertElementBefore("Inserted",3)
```

An iterator for the list is created...

```
SET I%ListIter=I%List.CreateIterator
```

Finally, a loop is used to traverse the list. Notice that the 4th and 5th elements were originally in the 3rd and 4th positions, respectively.

```
FOR QUIT:'I%ListIter.More DO $ENV.Output(I%ListIter.Next)
```

Iterating a Dictionary

Dictionaries are ordered by the value of a common property shared by all elements inserted into the Dictionary. This common property is known as the dictionary's **Key** property. The dictionary issues a **Watch** on the key property of all elements. Any time an element's key property changes, the dictionary receives a callback, and updates itself accordingly.

The following lines are intended to be entered into the Xecute shell, or an object browser. They may also be typed into a scratch method.

A **Dictionary** is created to store **Date** objects by their **TextMonth** property.

```
CREATE I%Dict=Base$Dictionary("TextMonth")
```

Next, **Date** objects are created for a number of different day values, beginning with the current day. Each **Date** object is inserted into the dictionary.

```
FOR T%Day=$H:-4567:1 CREATE T%Date=Base$Date(T%Day) DO I%Dict.InsertElement(T%Date)
```

An **Iterator** is created for the dictionary.

```
SET I%DictIter=I%Dict.CreateIterator
```

The iterator is used to traverse all Dictionary elements. The **TextDate** property of each element is placed in the output window.

```
FOR QUIT:'I%DictIter.More DO $ENV.Output(I%DictIter.Next.TextDate)
```

Iterating a Log

Logs order their elements based on a date/time value associated with each element. Elements can be inserted for the current time, or at some specific time in the past or future. For example, a Log could be used to keep track of events in the order that they occur, or it could be used to sort a number of elements with associated dates (such as date of birth) in chronological order.

In the following example, a Log is created...

```
CREATE I%TimeLog=Base$Log
```

Next, a time stamp object is created to specify a time at which the elements will be inserted...

```
CREATE I%Before=Base$TimeStamp
```

The **HANG** command is used to make certain that at least one second goes by before the next command is executed...

```
HANG 1
```

The first element is inserted at the current date and time.

```
DO I%TimeLog.InsertElement("Inserted First")
```

Next, a second element is inserted at the earlier date and time.

```
DO I%TimeLog.InsertElement("Inserted Second",I%Before)
```

An iterator for the log is created...

```
SET I%LogIter=I%TimeLog.CreateIterator
```

Finally, a loop is used to traverse the array. Notice that, since the fifth cell is empty, it will return "" as its value.

```
FOR QUIT:'I%LogIter.More DO $ENV.Output(I%LogIter.Next)
```

Iterators and Collections that Change

Unordered Collections

When traversing an *unordered* collection (Set or Bag) using an iterator, the iterator object obeys a contract which is order independent. The elements in the collection will be returned in an arbitrary order. Any items inserted into the collection before the iterator reaches the end are guaranteed to be visited, at some point.

This makes iterators complex. Suppose that two iterators, F and B, are traversing a Set collection. F is moving forward, and B is moving backward. At some time before either iterator has reached the end, a new element X is inserted into the Set. If both iterators continue undisturbed to the end of the collection, and element X is not removed, then both iterators are guaranteed to eventually reach element X. The following example illustrates this.

```
CREATE I%Set=Base$Set("Red","Orange","Yellow","Green","Blue","Violet")
```

A set is created, containing six of the seven colors of the rainbow.

```
SET I%F=I%Set.CreateIterator
SET I%B=I%Set.CreateIterator
SET I%B.IterationOrder="B"
```

Two iterators, **F** and **B**, are created to iterate the set. **F** will move **Forward**, and **B** will move **Backward**.

```
DO $ENV.Output(I%F.Next_ / "_I%B.Next)
DO $ENV.Output(I%F.Next_ / "_I%B.Next)
```

This shows the first two elements, in both the forward and backward directions.

```
DO I%Set.InsertElement("Indigo")
DO $ENV.Output("-----")
```

The seventh color, **Indigo**, is finally added.

```
FOR QUIT:'I%F.More DO $ENV.Output(I%F.Next_ / "_I%B.Next)
```

The remaining elements are displayed, both forward and backward. Note that **Indigo** is encountered in both directions. (Remember, the **Set** collection is *unordered*.)

Ordered Collections

An *ordered* collection makes certain promises about the order of items—in a List, the items are arranged according to sequential numeric positions. Iterators may or may not traverse newly-added elements. The iterator forms a contract that the elements will be visited in a certain order; thus it is not possible for it to visit newly inserted elements that have already been passed.

This example is similar in concept to the above example for unordered collections. Instead, however, a List is used. Because the List is ordered, the iteration process is different.

```
CREATE I%List=Base$List("Red","Orange","Yellow","Green","Blue","Violet")
```

A list is created, containing six of the seven colors of the rainbow.

```
SET I%F=I%List.CreateIterator
SET I%B=I%List.CreateIterator
SET I%B.IterationOrder="B"
```

Two iterators, **F** and **B**, are created to iterate the list. **F** will move **Forward**, and **B** will move **Backward**.

```
DO $ENV.Output(I%F.Next_ / "_I%B.Next)
DO $ENV.Output(I%F.Next_ / "_I%B.Next)
```

This shows the first two elements, in both the forward and backward directions.

```
DO I%List.InsertElement("Indigo")
DO $ENV.Output("-----")
```

The seventh color, *Indigo*, is finally added to the **List**.

```
FOR QUIT:'I%F.More DO $ENV.Output(I%F.Next_ / "_I%B.Next)
```

The remaining elements are displayed, both forward and backward. Note that *Indigo* is encountered in the forward direction, but not backward. (Remember, the *List* collection is *ordered*.)

Collection Operations

Deleting all Objects in a Collection

To delete all objects in a collection, without deleting the collection, use the iteration techniques described previously to get to each element. If the element is an object, you can send it a **DESTROY** message. In any case, you invoke the **RemoveElementAt** method to remove the element from the location in which it is stored.

Sending a DESTROY message to a collection destroys the collection. However, if the elements are objects, the objects are not deleted physically from the system. The DESTROY command has no effect on the objects contained in the collection. Only references to the objects in the collection are removed. Physical destruction only can occur by referencing the object directly and by sending it a DESTROY message.

The commands in these examples can be entered into the Xecute shell, or an object browser, or into a scratch method.

Case I: Also Destroying the Collection

First, we'll create a **Set** collection, containing a couple of non-object values.

```
CREATE I%TheSet=Base$Set(99,"Hello",32.765,"")
```

Next, we'll insert five objects into the set.

```
FOR T%X=1:1:5 CREATE T%Obj=ESI$Object DO I%TheSet.InsertElement(T%Obj)
```

Next, we'll create an **Iterator** for the collection.

```
SET I%Iter=I%TheSet.CreateIterator
```

Just to verify, we'll now examine the collection's contents.


```
FOR QUIT:'I%Iter.More SET T%Obj=I%Iter.Next DO $ENV.Output(T%Obj)
```

We'll now illustrate the process of destroying a collection along with all the elements it contains. The following **FOR** loop iterates the collection, destroying every object it contains.

```
DO I%Iter.Reset
FOR QUIT:'I%Iter.More SET T%Obj=I%Iter.Next IF $exist(T%Obj) DESTROY T%Obj
```

Notice that collection elements that are not objects, never need to be destroyed. Hence the use of the **\$EXIST** function above.

Finally, the collection itself is destroyed.

```
DESTROY I%TheSet
```

Note that it was *not* necessary to remove the elements from the collection, since the collection was being destroyed anyway.

Case II: The Collection is Not Destroyed

Just as above, we'll create a **Set** containing five objects as well as some non-object values, and an **Iterator** for the collection.

```
CREATE I%TheSet=Base$Set(99,"Hello",32.765,"")
FOR T%X=1:1:5 CREATE T%Obj=ESI$Object DO I%TheSet.InsertElement(T%Obj)
SET I%Iter=I%TheSet.CreateIterator
```

Just to verify, we'll examine the collection's elements.

```
FOR QUIT:'I%Iter.More SET T%Obj=I%Iter.Next DO $ENV.Output(T%Obj)
```

Next, all elements in the collection are destroyed. (See **Case I** above for an explanation.)

```
DO I%Iter.Reset
FOR QUIT:'I%Iter.More SET T%Obj=I%Iter.Next IF $exist(T%Obj) DESTROY T%Obj
```

Finally, all collection elements are removed.

```
DO I%TheSet.RemoveAll
```

Case III: Remove and Destroy Objects Only

In this case, the collection is not destroyed. Furthermore, only objects need to be removed from the collection—any non-object value does not need to be removed. The objects being removed also need to be destroyed.

We'll begin by using the same setup lines as in the above examples.

```
CREATE I%TheSet=Base$Set(99,"Hello",32.765,"")
FOR T%X=1:1:5 CREATE T%Obj=ESI$Object DO I%TheSet.InsertElement(T%Obj)
SET I%Iter=I%TheSet.CreateIterator
```

Before going on, we can examine the contents of the collection.

```
FOR QUIT:'I%Iter.More SET T%Elem=I%Iter.Next DO $ENV.Output(T%Elem)
```

Now the elements that are objects are removed and destroyed, while the other elements are left intact.

```
DO I%Iter.Reset
FOR QUIT:'I%Iter.More SET T%Obj=I%Iter.Next IF $exist(T%Obj) DO I%TheSet.RemoveElementAt(I%Iter) DESTROY T%Obj
DO $ENV.Output("-----")
```

Now we'll look at the collection's contents again, making sure that only the objects were all removed.

```
DO I%Iter.Reset
FOR QUIT:'I%Iter.More SET T%Elem=I%Iter.Next DO $ENV.Output(T%Elem)
```

Creating a "Stack" Collection

EsiObjects does not include a **Stack** collection class. However, it is possible to use the **List** collection to implement the behavior of a last-in/first-out (LIFO) stack. The following methods are used to achieve this behavior:

InsertFirstElement used to add a new item to the head of the list

RetrieveFirstElement used to obtain the value of the element at the head of the list

RemoveFirstElement used to delete the element at the head of the list

IsEmpty used to determine whether any items remain in the list

Thus, using a List, the classic Stack "Push" operation would be equivalent to InsertFirstElement, while the classic Stack "Pop" operation would be equivalent to the combination of RetrieveFirstElement and RemoveFirstElement.

The following three lines of code may be typed into a method, or entered from either the Xecute shell or an object browser. First, a list is created to emulate a stack.

```
CREATE I%Stack=Base$List
```

Next, five items are "pushed" onto the stack using InsertFirstElement.

```
FOR T%X="first","second","third","fourth","fifth" DO I%Stack.InsertFirstElement(T%X)
```

Finally, the items are "popped" from the stack using the combination of RetrieveFirstElement to obtain the value of each item, and RemoveFirstElement to delete the item. Each item is placed in the output window.

```
FOR QUIT:I%Stack.IsEmpty SET T%Item=I%Stack.RetrieveFirstElement DO I%Stack.RemoveFirstElement,$ENV.Output(T%Item)
```

Notice that the items appear in the reverse of the order in which they were inserted: they come out *backwards*. This is entirely in keeping with the LIFO order of stacks.

Creating a "Queue" Collection

EsiObjects does not include a **Queue** collection class. However, it is possible to use the **List** collection to implement the behavior of a first-in/first-out (FIFO) queue. The following methods are used to achieve this behavior:

InsertLastElement used to add a new item to the tail of the list

RetrieveFirstElement used to obtain the value of the element at the head of the list

RemoveFirstElement used to delete the element at the head of the list

IsEmpty used to determine whether any items remain in the list

Thus, using a List, the classic Queue "EnQueue" operation would be equivalent to InsertLastElement, while the classic Queue "DeQueue" operation would be equivalent to the combination of RetrieveFirstElement and RemoveFirstElement.

The following three lines of code may be typed into a method, or entered from either the Xecute shell or an object browser. First, a list is created to emulate a queue.

```
CREATE I%Queue=Base$List
```

Next, five items are "enqueued" onto the stack using InsertLastElement.

```
FOR T%X="first","second","third","fourth","fifth" DO I%Queue.InsertLastElement(T%X)
```

Finally, the items are "dequeued" from the queue using the combination of RetrieveFirstElement to obtain the value of each item, and RemoveFirstElement to delete the item. Each item is placed in the output window.

```
FOR QUIT:I%Queue.IsEmpty SET T%Item=I%Queue.RetrieveFirstElement DO I%Queue.RemoveFirstElement,$ENV.Output(T%Item)
```

Notice that the items appear in the same order in which they were inserted: they come out *forwards*. This is entirely in keeping with the FIFO order of queues.

Using Immutable Classes

What are Immutable Classes?

Immutable classes are classes that produce instances whose values cannot change. Date and time values are classically immutable. For example, a persons birth date and time are immutable.

For a detailed discussion of virtual objects read the section at this path: [Using Objects, Building Objects, Virtual Objects](#).

Immutable classes supplied with EsiObjects can be found in the **Base** library under the abstract class [Immutable](#). They are:

- Date
- Interval
- MVariable
- NameValuePair
- TimeRange
- TimeStamp

To learn the details of the services each class provides, use the Session Browser to migrate the structures. Make sure you Documentation Window is visible.

Immutable Protocols

Immutable Hierarchy

Immutable Class

Sets are unordered collections (similar to the Bag collection) that do not allow duplicates.

Date Class

Sets are unordered collections (similar to the Bag collection) that do not allow duplicates.

Interval Class

Sets are unordered collections (similar to the Bag collection) that do not allow duplicates.

Mvariable Class

Sets are unordered collections (similar to the Bag collection) that do not allow duplicates.

NameValuePair Class

Sets are unordered collections (similar to the Bag collection) that do not allow duplicates.

Time Class

Sets are unordered collections (similar to the Bag collection) that do not allow duplicates.

TimeRange Class

Sets are unordered collections (similar to the Bag collection) that do not allow duplicates.

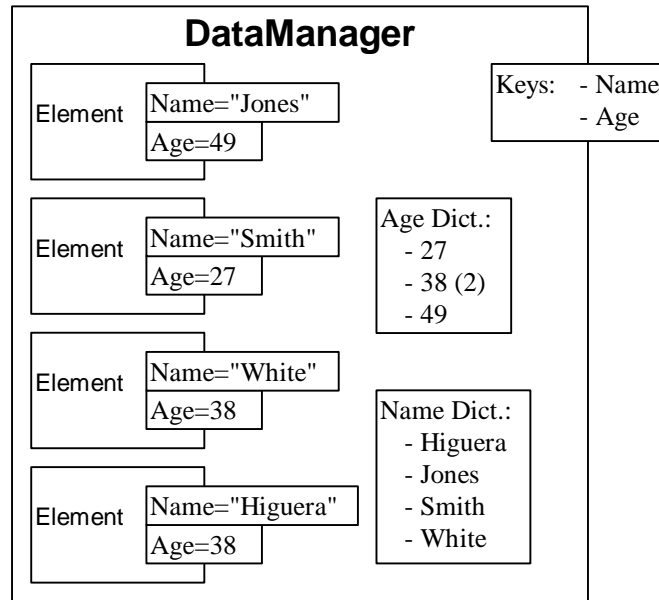
TimeStamp Class

Sets are unordered collections (similar to the Bag collection) that do not allow duplicates.

Using the DataManager Class

What is a DataManager Class?

The concrete base class `DataManager` is an aggregate object containing unique, homogeneous objects arranged according to zero or more key property values.



Two other collection classes are used to implement the `DataManager`. They are:

- **Dictionary** is collection class used to organize key property values. Dictionaries have the unique ability to maintain themselves when an object it knows about is altered. `EsiObjects` event handling makes this possible.
- **Set** is used to maintain the master index of objects the `DataManager` knows about. A `Set` is used since the `DataManager`'s elements are unique.

`DataManager` is used for complex organization of homogeneous data in ways that would not be possible with a simple `Dictionary`, or other collection. The objects in a `DataManager` are assumed to be of the same class, but this is loosely enforced. However, the objects *must* be homogeneous. For example, it is technically possible to insert a number of different collection objects (i.e. `Bag`, `Set`, `List`, `Array`, `Dictionary` and `Log`), as long as the `DataManager` will only use them in ways that are appropriate for any kind of `Collection`.

Creating and Destroying a DataManager Object

Creating a DataManager

The **CREATE** command supports a creation parameter for the `Class` property. This property can only be defined at runtime, so it is required. Also, note that the `Share`

creation keyword can be used to specify whether the DataManager is persistent (1) or transient (0). In order to be available across multiple EsiObjects sessions, the DataManager must be persistent.

Example

The following CREATE command creates a DataManager in the instance variable DM. The DataManager is persistent, and the objects inserted into it will be of class "Patient."

```
CREATE I%DM=Base$DataManager:Share=1:(Class="Patient")
```

Destroying a DataManager

If data control is turned on, then DESTROY will specifically destroy all the items in the DataManager and flush all the dictionaries. Otherwise the data manager is simply destroyed as a normal object.

Example

```
DESTROY I%DM
```

The DataManager Interface

Class Property

Returns the name of the base class: items inserted into the data manager are presumed to be of this class. This class will be used whenever new items are created, and any items inserted into the DataManager must implement all of its key properties.

Access:

Create, Value

Format:

Class returns a string, the class name of the elements.

Examples:

Create:

```
CREATE I%DM=Base$DataManager:Share=1:(Class="Patient")
```

Value:

```
SET T%DMClassName=I%DM.Class
```

ControlsData Property

True if the items inserted into the data manager will be treated as components. In that case, the data manager will destroy all items when they are removed, or when the data

manager itself is destroyed. If ControlsData is false, then the data manager will treat its elements as non-component objects, and will not destroy them.

Access:

Assign, Value

Format:

ControlsData returns 1/0 (1 for true, 0 for false).

Example

Assign:

```
SET I%DM.ControlsData=1
```

Value:

```
IF 'I%DM.ControlsData DESTROY T%ThisElement
```

CreateElement Method

Creates a new element of the data manager's base class (defined by the Class property).

Input:

none.

Return value:

the newly created object.

Side effects:

the new object is also inserted into the data manager.

Example

```
SET T%NewItem=I%DM.CreateElement  
SET T%NewItem.Name="Jane Doe"
```

InsertElement Method

Adds an element to the Data Manager, which also causes it to be added to all the dictionaries in the dictionary list.

Input:

The element to be inserted.

Return:

none.

Side effects:

none.

Example

```
DO I%DM.InsertElement(T%ThisPatient)
```

RemoveElement Method

Removes the specified element from the data manager (and its associated dictionaries). Destroys the element if data control is turned on.

Input:

The specific element to be removed.

Returns:

none.

Side effects:

The element may also be destroyed, if instance control is turned on.

Usage:

This method is invoked when a specific element needs to be removed from the data manager. It is not a way to find and remove a specific element. Finding elements requires the SelectMatches method.

Example

```
DO I%DM.RemoveElement(T%DelPat)
```

Cardinality Property

Returns the number of elements in the data manager.

Access:

Value

Format:

Cardinality returns a non-negative integer; the number of elements in the data manager.

SelectMatches Method

Returns the set of elements matching the specified criteria.

Input:

Criteria to be applied to each element in the Data Manager. The input value is a single criterion; note that if complex criteria need to be satisfied, then simple criteria may be combined by using `AndCompoundCriteria` and `OrCompoundCriteria` to produce a single complex criterion.

Return Value:

A set of matching elements.

Side effects:

A set object is created, containing the elements that match the criteria.

Example

```
SET T%Matches=I%DM.SelectMatches(T%FemalesOver65)
```

Keys Property

An array of keys suitable for \$ordering. Subscripted by key property name.

Access:

\$order

Format:

Keys(propertyname)

Example

```
SET T%Prop=""
FOR SET T%Prop=$order(I%DM.Keys(T%Prop)) QUIT:T%Prop="" DO T%KeyList.AddElement(T%Prop)
```

AddKey Method

Adds a new key property to the data manager. If the specified property is not already being tracked, it will create a new dictionary for that property and copy the items into that new dictionary.

RemoveKey Method

Removes a key from the data manager's list of key properties.

Input:

The key property name to remove.

Return:

none.

Side effects:

none.

Example

```
DO I%DM.RemoveKey( "SSN" )
```

Using Criteria Classes

What are Criteria Classes?

Each **Criteria** member class represents a true/false criterion, determining whether or not some object matches a certain requirement (or possibly a set of requirements.) This class and its descendants were designed for windowing, to be repeatedly applied to each object in a group such as a DataManager, in order to evaluate which of them meet a given set of requirements.

Member Classes:

RangeCriteria ; Falls within a certain range

FilterCriteria ; General Filtering

Related Classes:

DataManager ; Contains objects to evaluate using criteria.

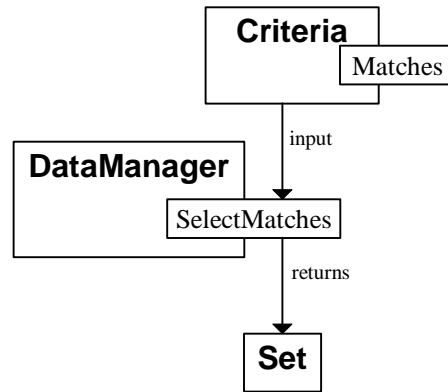
Interfaces:

Primary

IsRange	True if the criterion is a RangeCriteria, false if not.
Matches	Tests a given object to see whether it meets the criterion.
Properties	Returns the object properties that must be implemented for the criterion to be applied to the object.

Usage:

Criteria are frequently used together with the DataManager class. DataManager implements a method called SelectMatches, which accepts any single criterion as input.



In the above diagram, the `SelectMatches` method of a `DataManager` object is being invoked, and a `Criteria` object is being passed to it as input. The `SelectMatches` method returns a `Set` containing the elements (if any) that match the specified criterion.

Example

Let's suppose that we have a `DataManager` object in the instance variable `DM`, and that the `Patient` objects in this `DataManager` implement two relevant properties, `Sex` ("M" or "F") and `Age` (integer.) Let's further assume that our intent is to obtain a collection containing all the female patients over the age of 65. An `ExactHitCriteria` object is used for the sex, a `RangeCriteria` for the age, and an `AndCompoundCriteria` is used to combine the two.

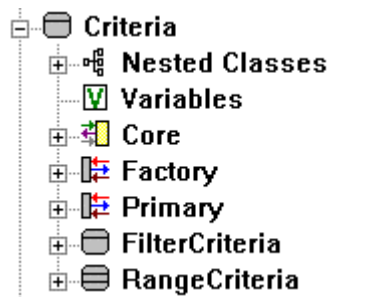
```

CREATE T%Female=Base$ExactHitCriteria("Sex","F")
CREATE T%Over65=Base$RangeCriteria("Age",65,999)
CREATE T%FemaleOver65=Base$AndCompoundCriteria
DO T%FemaleOver65.AddCriteria(T%Female,T%Over65)
SET I%MatchList=I%DM.SelectMatches(T%FemaleOver65)
  
```

Criteria Protocol

Criteria Hierarchy

This section describes the protocol of each class in the **Criteria** hierarchy. The class hierarchy of the criteria classes is shown in the following figure.



The root class of all Criteria classes is the Criteria class. The class hierarchy is located in the Base Library. The two subclasses of Criteria are:

- FilterCriteria
- RangeCriteria

The following sections summarize the properties and methods for each Criteria class.

Criteria Class

The Criteria class is an abstract class. No instances of this class can be created. The class is simply a placeholder for data and methods that can be implemented or shared by its subclasses.

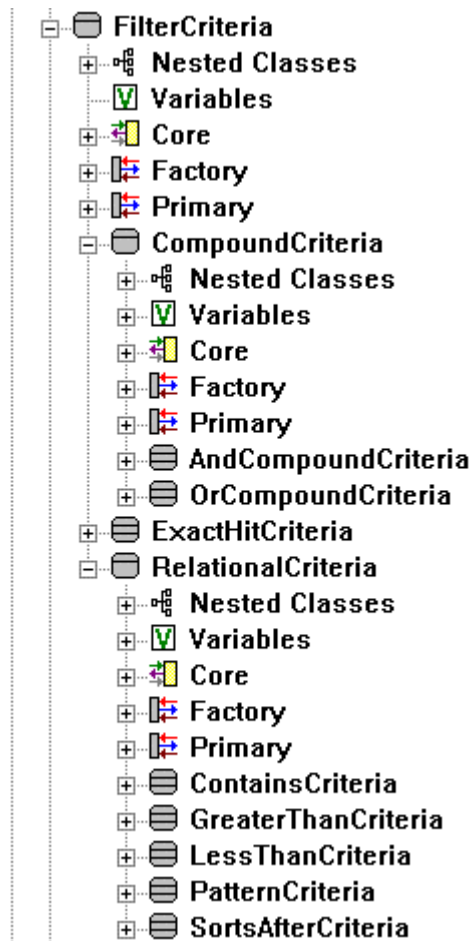
The following is a summary of the properties and methods that are part of the Criteria protocol. Many of these methods have no functionality and are intended to be overridden by the subclasses.

FilterCriteria

The FilterCriteria class is an abstract class that provides no services to the hierarchy. It is a placeholder for its subclasses which are:

- CompoundCriteria
- ExactHitCriteria
- RelationalCriteria

CompoundCriteria and ExactHitCriteria are concrete classes that implement services. RelationalCriteria is an abstract class that specializes the FilterCriteria for relational criteria checks.



A general grouping of Criteria classes specifically devoted to filtering objects. Each object is evaluated according to a specific criterion (or compound set of criteria).

Methods

Matches Returns true if the specified object matches the criteria.

Input: An object, to which to apply the criterion.

Return value: 1/0 (true if the object matches the criterion, false if it does not or the criterion is not fully specified.)

Side effects: None.

Example

```
IF T%ThisCriter.Matches(T%ThisObject) DO T%List.InsertElement(T%
ThisObject)
```

Properties

Properties Returns a collection containing the properties affected by the criterion. An object must implement these properties in order to be used by the

Matches method for this class.

Access: Value

Subscripts: None.

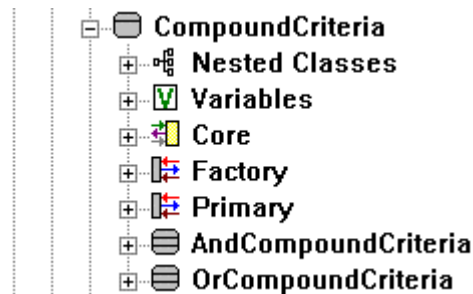
IsRange Returns true if the criteria are based on a range (exposing RangeStart and RangeEnd properties), false if not. Returns false in this case.

Access: Value

Subscripts: None.

CompoundCriteria Class

The CompoundCriteria class is an abstract class that has two subclasses shown in the diagram below.



A compound criterion contains multiple component criteria, and returns true or false based on the truth or falsity of its component criteria.

DESTROYing a CompoundCriteria explicitly destroys all of its individual component criteria.

This abstract class implements several services inherited by its subclasses.

Methods

Matches Returns true if the specified object satisfies the criteria.

Input: An object, to which to apply the component criteria.

Return value: 1/0 (true if the object matches the criterion, false if it does not or if the criterion is not fully specified.)

Side effects: None.

Example

```
IF T%ThisCriter.Matches(T%ThisObject) DO T%List.InsertElement(T%ThisObject)
```

AddCriteria Adds one or more new criteria as a component of the CompoundCriteria.

Input: The criteria to be added (multiple).

Return value: None.

Side effects: None.

RemoveCriteria Removes one or more criteria from this compound criteria.

Input: The criteria to remove. (multiple)

Return Value: None.

Side effects: None.

Properties

Properties

Returns a collection containing the properties affected by the criterion. An object must implement these properties in order to be used by the Matches method for this class. Since a CompoundCriteria may contain any number of individual Criteria, many different properties may be referenced by it.

Access: Value

Subscripts: None.

IsRange

Returns true if the criteria are based on a range (exposing RangeStart and RangeEnd properties), false if not. Returns false in this case.

Access: Value

Subscripts: None.

AndCompoundCriteria Class

Overview

A conjunctive compound criterion, returning TRUE when all of its component criteria are true, and FALSE if any of them is not true (or if it does not contain any criteria.)

DESTROYing an AndCompoundCriteria explicitly destroys all of its component criteria.

Methods

AddCriteria Adds one or more new criteria as a component of the AndCompoundCriteria.

Input: The criteria to be added (multiple).

Return value: None.

Side effects: None.

RemoveCriteria Removes one or more criteria from this AndCompoundCriteria.

Input: The criteria to remove. (multiple)

Return Value: None.

Side effects: None.

Matches Returns true if the ALL of the specified criteria are true, or false if any are false. Also returns false if there are no criteria.

Input: An object, to which to apply the criterion.

Return value: 1/0 (true if the object matches the criteria, false if it does not or no component criteria are specified.)

Side effects: None.

Example

```
IF T%ThisCriter.Matches(T%ThisObject) DO T%List.InsertElement(T%ThisObject)
```

Properties

IsRange Returns true if the criteria are based on a range (exposing RangeStart and RangeEnd properties), false if not. Returns false in this case.

Access: Value

Subscripts: None.

Properties Returns a set containing the properties affected by the criterion. An object must implement these properties in order to be used by the Matches method for this class. Since an AndCompoundCriteria may contain any number of individual criteria, many different properties may be referenced by it.

Access: Value

Subscripts: None.

OrCompoundCriteria Class

Overview

A disjunctive compound criterion, returning TRUE when any of its component criteria are true, and FALSE if all of them are false (or if it does not contain any criteria.)

DESTROYing an OrCompoundCriteria explicitly destroys all of its component criteria.

Properties

IsRange Returns true if the criteria are based on a range (exposing RangeStart and RangeEnd properties), false if not. Returns false in this case.

Access: Value

Subscripts: None.

Properties Returns a collection containing the properties affected by the criterion. An object must implement these properties in order to be used by the Matches method for this class. Since a CompoundCriteria may contain any number of individual criteria, many different properties may be referenced by it.

Access: Value

Subscripts: None.

Methods

AddCriteria Adds one or more new criteria as a component of the OrCompoundCriteria.

Input: The criteria to be added (multiple).

Return value: None.

Side effects: None.

RemoveCriteria Removes one or more criteria from this OrCompoundCriteria.

Input: The criteria to remove. (multiple)

Return Value: None.

Side effects: None.

Matches Returns true if ANY of the component criteria are true, false only if ALL are false. Also returns false if there are no component criteria.

Input: An object, to which to apply the criterion.

Return value: 1/0 (true if the object matches the criterion, false if it does not or if the criterion is not fully specified.)

Side effects: None.

Example

```
IF T%ThisCriter.Matches(T%ThisObject) DO T%List.InsertElement(T%ThisObject)
```

ExactHitCriteria Class

Overview

This criterion returns true if an object's property is exactly equal to a certain value. The services are described below.

Methods

Matches Returns true if a certain property of the specified object is equal to a certain value. The certain property is defined by the ExactHitCriteria's Property property, and the certain value is defined by the ExactHitCriteria's Value property.

Input: An object, to which to apply the criterion.

Return value: 1/0 (true if the object matches the criterion, false if it does not or if the criterion is not fully specified.)

Side effects: None.

Example

```
IF T%ThisCriter.Matches(T%ThisObject) DO T%List.InsertElement(T%ThisObject)
```

Properties

Property The property affected by the ExactHitCriteria. The Matches method will reference this property to determine whether it equals the value.

Access: Assign, Value

Subscripts: None.

Value The exact value that will be compared to an object's property by the Matches method.

Access: Assign, Value

Subscripts: None.

Properties Returns a collection containing the properties affected by the criterion. An object must implement these properties in order to be used by the Matches method for this class.

Access: Value

Subscripts: None.

IsRange Returns true if the criteria are based on a range (exposing

RangeStart and RangeEnd properties), false if not. Returns false in this case.

Access: Value.

Subscripts: None.

RelationalCriteria

ContainsCriteria

GreaterThanCriteria

LessThanCriteria

PatternCriteria Class

Overview

A criteria that returns true if a specified object's property matches a certain pattern, or false if it fails to match (or no pattern is defined.)

CREATE Command

Creates a PatternCriteria object.

Input:

Property

Pattern

Example

```
CREATE T%StateAbbrev=Base$PatternCriteria("State", "2U")
```

Methods

Matches Returns true if the value of the specified object's property matches the specified pattern. Returns false if the property does not match the pattern, or if no pattern is defined.

Input: An object, to which to apply the criterion.

Return value: 1/0 (true if the object matches the criterion, false if it does not or if the criterion is not fully specified.)

Side effects: None.

Example

```
IF T%ThisCriter.Matches(T%ThisObject) DO T%List.InsertElement(T%ThisObject)
```

Properties

Pattern A pattern to be applied by the Matches method.

Access: Assign, Value

Subscripts: None.

Property Returns the property affected by the criteria. This property will be referenced when an object is passed to the Matches method.

Access: Assign, Value

Subscripts: None.

Properties Returns a collection containing the properties affected by the criterion. An object must implement these properties in order to be used by the Matches method for this class.

Access: Value

Subscripts: None.

IsRange Returns true if the criteria are based on a range (exposing RangeStart and RangeEnd properties), false if not. Returns false in this case.

Access: Value

Subscripts: None.

RangeCriteria Class

RangeCriteria determines whether a specific property of an object falls between a pre-defined starting and ending value (non-inclusive).

CREATE Command

Creates a range criteria object.

Input:

Property

Start

End

Methods

Matches

Returns true if the value of the specified object's range property is between the RangeStart and RangeEnd properties (non-inclusive).

Input: An object, to which to apply the criterion.

Return value: 1/0 (true if the object matches the criterion, false if it does not or if the criterion is not fully specified.)

Side effects: None.

Example

```
IF T%ThisCriter.Matches(T%ThisObject) DO T%List.InsertElement(T%ThisObject)
```

Properties

IsRange

Returns true if the criterion is based on range (exposing RangeStart and RangeEnd properties), false if not. Always true for RangeCriteria.

Access: Value

Subscripts: None.

Properties

Returns a collection containing the properties affected by the criterion. An object must implement these properties in order to be used by the Matches method for this class.

Access: Value

Subscripts: None.

Property

Returns the property affected by the criteria. This property will be referenced when an object is passed to the Matches method.

Access: Assign, Value

Subscripts: None.

RangeEnd

Returns the range's non-inclusive ending value.

Access: Assign, Value

Subscripts: None.

RangeStart

Returns the range's non-inclusive starting value.

Access: Assign, Value

Subscripts: None.

Using Mix-In Classes

What are Mix-In Classes?

Mix-in classes are abstract classes that are linked into some point of the class hierarchy where the set of services they contain can be inherited by all of its descendants through multiple inheritance. They can hold all definitional components of an abstract class. For example, the mix-in class `Base$AbsLockableObject` can be linked in at any level of a class hierarchy. At that point, all database locking services it offers are inherited by its descendants.

Adding Interfaces Using Mix-In Classes

The general user of an object uses a simplified view of an object. This view is presented through the primary interface. Each object can have any number of different interfaces. Each interface has a specific purpose and audience.

EsiObjects has a direct syntactic support of interfaces. It is important to know the difference between an EsiObjects interface and the abstraction notion of an interface. An EsiObjects interface consists of a namespace of services (events, properties, methods and relationships) supported in a class. Each class can have multiple EsiObjects interfaces.

The abstract interface is the sum of all services supported by a class and is the combination of all EsiObjects interfaces for that class.

The term interface is often used to refer to a specific set of services and not necessarily all the services. This can make the term interface confusing. To avoid confusion, the term *abstract interface* is used to refer to the entire set of services supported by an object. The term *protocol group* is used to refer to a specific subset of services. The term interface refers to the EsiObjects interface.

Interfaces provide a simple mechanism for controlling the complexity of an object. They allow the developer of the object partition services into private and public groups. Public services are generally made available to the object user through the Primary interface. All other interfaces are generally internal or private to the object.

Accessing Interfaces

When working with object services, the syntax of the service name is as follows:

Object.[Interface::]Method

Object. [Interface::]Property

Object. [Interface::]Event

Object. [Interface::]Relationship

If the interface name is omitted, then the current default interface name is used (the Primary interface).

In the following example, the internal state is verified by using the validate method found in the factory interface:

```
;
IF 'T%Room.Factory::Validate DO $ENV.Assert("Invalid")
;
```

Major Interface

List of Interfaces

By default, whenever a class is created, the **Primary** interface is automatically created. It is used to expose all external or public services to the object user.


The **Factory** interface can be added to the class by the programmer. It generally contains private services that are used in the creation, validation and deletion of instances of the class. The Factory interface may contain methods that have reserved names, namely, CREATE, DESTROY, InitSysVars and InitClassVars. CREATE and DESTROY may be created by you. They must be spelled correctly and totally in uppercase. InitSysVars and InitClassVars are reserved and are created by the system. You cannot use these names.

The Primary and Factory names are reserved by EsiObjects. They must be spelled correctly when used.

Other interfaces can be added to a class either by adding it directly or by linking in a Mix-In class. EsiObjects contains some common Mix-In classes in the Base library. Some of these classes are complete and others act a templates where you override the interface and services with a specific implementation.

These Mix-In classes are provided by EsiObjects in the Base library:

- **Attachment** — append information to an object.
- **Lockable** — add database locking services.
- **Security** — control object access.
- **Serialization** — save or restore object from serial media.

You can also create your own Mix-In classes. They are no different from a normal class. They simply have a property that identifies them as Mix-In. They are visually identifiable as Mix-In by this icon: 

Primary Interface

The Primary interface is the main interface to an object and is automatically created when the class is created. It must be spelled correctly and contains the public services you want to expose to the object user.

Factory Interface

The factory interface may assist in the the life-cycle services needed to create, maintain and destroy object of the class that inherits the interface. The class used to add the factory interface to your class is **AbsFactoryObject**. It adds an interface called Factory.

Once you link the AbsFactoryObject into your class, you can override the interface in your class and add the services you need. The following table describes the type of services that can be added.

Type of Service	Description	Service	Service Type
Object Creation	If the CREATE method exists, it is invoked by the system when you use the Create command. It is used to assist in the creation of the object. It must be spelled in uppercase.	CREATE	System method
Object Destruction	If the DESTROY method exists, it is invoked by the system when the user invokes the Destroy command. It is used to clean up behind the object. The ObjectDead event is automatically invoked by the system when the object dies. It can be watched by other objects.	DESTROY ObjectDead	System Method Event
Identification	Used to access name and identify information.	ID Name Class ClassName Domain	Property (R) Property (RW) Property (R) Property (R) Property (R)
Validation	Used to verify or validate contents of the object.	Validate	Method

Referencing	Provide reference counting services. When an object is created, it is automatically given a reference count of 1. This can be accessed via the \$Reference special variable. When it is destroyed via the Destroy command, the objects reference count is decremented by one. If it is less than one, the object is literally destroyed. The Preserve command can be used to increment the reference count in order to keep that object alive. This capability is useful when objects are used in a multi-user environment.	AddRef ReleaseRef	Method Method
Browsing	Find internal state information.	CopyInstanceTable	Method

Security Interface

The **Security** interface can be used to control access to an object. The base class for the security interface is **AbsSecurityObject**.

The following table describes some of the types of services that can be offered by the Security interface.

Type of Service	Description	Service	Service Type
Secure	Used to Change the security state of the object.	Secure	Method
Validate Access	Validate destruction and clean up the object.	VerifyAccess	Method
Information	Find information about the access requirements to an object.	GetACL	Method
Initialization	Create initial security information.	CREATE	Method
Cleanup	Clean up and security information.	DESTROY	Method

Serialization Interface

The **Serialization** interface allows objects to be saved and restored from serial media. The base class for the serialization interface is **AbsSerilizationObject**.

The following table describes some of the services that may be offered by the Serialization interface.

Type of Service	Description	Service	Service Type
Object Interchange Input	Restore the object from a serial source in the EsiObjects Interchange format.	RestoreFormatted Object	Method
Object Interchange Output	Save the object into a serial destination in the EsiObjects Object Interchange format.	SaveFormatted	Method
Diagnostic output	Present a diagnostic serial dump of an object.	Dump	Method

Attachment Interface

The **Attachment** interface allows other objects to attach data to the object. This allows the object to carry the baggage that can be associated with it from other contexts. The base class for the Attachment interface is **AbsAttachmentObject**.

The following table describes the services offered by the attachment interface.

Type of Service	Description	Service	Service Type
Object Specific	Local to calling object.	Object	Property (RWD)
Context	Shared by token.	Token	Property (RWD)
Class	Shared across class.	Class	Property (RWD)
Cleanup	Clean up the associated store.	DESTROY	Method

Using the XML Parser

The EsiObjects XML parser conforms to the SAX2 specification. It is not a full implementation of this specification.

Overview

Base library components are:

1. Base\$XMLReaderImpl,
2. Base\$AttributesImpl, and
3. Base\$XMLContentHandler

Interface

A Simple Example

Part 4: External Interface

External Call Interface

EsiObjects can access legacy M code and data just as any M program can. Additionally, legacy M code can access EsiObjects code via a program interface known as the **External Call Interface**.

Many times you may have written an object to do some work, but need to access that object from legacy M code. The API contained in the routine VESOEX allows you to do this.

By invoking various tags and passing in the proper parameters to these tags, you can create and destroy objects, invoke methods and property accessors on those objects.

Initialization

To use the external call interface, in your M routine you must first initialize your M stack to be able to use the object services. This is done by calling the INIT tag.

```
DO INIT^VESOEX
```

No return value is returned and no inputs needed. This now initializes your M stack for object services.

The table below describes each service currently available in the API. Other services may be added in the future.

Service	Tag	# Inputs	Return Value
Create Object	CREATE^VESOEX	4	Object Id
Invoke method	INVOKE^VESOEX	3+	Method return
Set Property	SETPROP^VESOEX	4+	1 or 0
Get Property	GETPROP^VESOEX	3+	Property Value
Destroy Object	DESTROY^VESOEX	1	None
Initialize	INIT^VESOEX	0	None

API Inputs

Each set of input parameters for the tags is discussed below.

CREATE

```
CREATE^VESOEX(classname, inputs, object_type, .errcode)
```

classname: the fully qualified (library and class name) name of the class

inputs: an array for passing in parameters. The format of the array is as follows:

array=*n* (the number of input parameters)

array(1)=value of first parameter

.

.

.

array(n)=value of *nth* parameter

object_type: a code for what type of object being created,. For all except “S”, *object_type(1)* should be set to the appropriate global specification.

“S” = shared object

“F” = fixed location

“B” = base location

“C” = child object

errcode: not currently used

Example:

If we are creating a Patient object at a base location ^PAT, passing in an MRN on the CREATE, we would invoke the API as follows:

```
; Set the object to be based at ^PAT
Set objfix="B",objfix(1)="^PAT"
; Set the 1 input parameter
Set input=1,input(1)=123456
Set ObjId=$$CREATE^VESOEX("User$Patient",input,objfix,0)
```

INVOKE

INVOKE^VESOEX(*ObjId*,*MethodName*,*InputCnt*,*input_1*...*input_n*)

ObjId: Object Id

MethodName: the name of the method. If invoking a method in an interface other than Primary, use the syntax: *Interface::MethodName*

InputCnt: The number of parameters being passed into this method

input_1: The value of the first parameter

.

.

.

input_n: The value of *nth* parameter

Example:

If we were invoking the Admit method on the Patient passing in a Location code we would invoke the API as follows:

```
Set Value=$$INVOKE^VESOEX(ObjId,"Admit",1,"ER")
```

Note that ObjId was gotten from a previous CREATE call as described above.

SETPROP

```
SETPROP^VESOEX(ObjId,PropertyName,PropValue,InputCnt,input_1...input_n)
```

ObjId: Object Id

PropertyName: the name of the property. If invoking a property in an interface other than Primary, use the syntax: *Interface::PropertyName*

PropValue: the value of the property

InputCnt: The number of parameters being passed into this method

input_1:The value of the first parameter

.
.

.

input_n: The value of nth parameter

Example:

If we were assigning the DOB on the Patient we would invoke the API as follows:

```
Set RetValue=$$SETPROP^VESOEX(ObjId,"DOB","10/12/1960",0)
```

GETPROP

```
GETPROP^VESOEX(ObjId,,PropertyName,InputCnt,input_1...input_n)
```

ObjId: Object Id

PropertyName: the name of the property. If invoking a property in an interface other than Primary, use the syntax: *Interface::PropertyName*

InputCnt: The number of parameters being passed into this method

input_1:The value of the first parameter

.
.

.

input_n: The value of nth parameter

Example:

If we were getting the DOB on the Patient we would invoke the API as follows:

```
Set DOB=$GETPROP^VESOEX(ObjId,"DOB",0)
```

DESTROY

```
DESTROY^VESOEX(ObjId)
```

ObjId: Object Id

Example:

If we were destroying the Patient object we would invoke the API as follows:

```
Do DESTROY^VESOEX(ObjId)
```