Customizing Express Applications

iPlanet™ Unified Development Server

Version 5.0

Contents

List of Figures

List of Procedures

List of Code Examples

Preface

This manual describes the architecture of applications created by Express. Once you are familiar with Express architecture, you can begin to customize your applications.

This preface contains the following sections:

Product Name Change

Forte 4GL has been renamed the iPlanet Unified Development Server. You will see full references to this name, as well as the abbreviations iPlanet UDS and UDS.

Audience for This Guide

This manual is intended for application developers. We assume that you:

- have programming experience

- are familiar with SQL and your particular database management system

- understand the basic concepts of Express as described in *A Guide to Express*

- have an Express application that you want to customize

Organization of This Guide

The following table briefly describes the contents of each chapter:

Chapter	Description
Chapter 1, "Express Application Architecture"	Provides an overview of the architecture of Express applications.
Chapter 2, "Customizing Express Applications"	Provides a variety of customization examples and identifies others available online.
Appendix A, "Express Example Applications"	Express example applications.

Text Conventions

This section provides information about the conventions used in this document.

Format	Description
italics	Italicized text is used to designate a document title, for emphasis, or for a word or phrase being introduced.
`monospace`	Monospace text represents example code, commands that you enter on the command line, directory, file, or path names, error message text, class names, method names (including all elements in the signature), package names, reserved words, and URLs.

Format	Description
ALL CAPS	Text in all capitals represents environment variables (FORTE_ROOT) or acronyms (UDS, JSP, iMQ).
	Uppercase text can also represent a constant. Type uppercase text exactly as shown.
Key+Key	Simultaneous keystrokes are joined with a plus sign: Ctrl+A means press both keys simultaneously.
Key-Key	Consecutive keystrokes are joined with a hyphen: Esc-S means press the Esc key, release it, then press the S key.

Other Documentation Resources

In addition to this guide, there are additional documentation resources, which are listed in the following sections. The documentation for all iPlanet UDS products (including Express, WebEnterprise, and WebEnterprise Designer) can be found on the iPlanet UDS Documentation CD. Be sure to read "Viewing and Searching PDF Files" on page 19 to learn how to view and search the documentation on the iPlanet UDS Documentation CD.

iPlanet UDS documentation can also be found online at http://docs.iplanet.com/docs/manuals/uds.html.

The titles of the iPlanet UDS documentation are listed in the following sections.

iPlanet UDS Documentation

- *A Guide to the iPlanet UDS Workshops*

- *Accessing Databases*

- *Building International Applications*

- *Escript and System Agent Reference Guide*

- *Fscript Reference Guide*

- *Getting Started With iPlanet UDS*

- *Integrating with External Systems*

- *iPlanet UDS Java Interoperability Guide*

- *iPlanet UDS Programming Guide*

- *iPlanet UDS System Installation Guide*

- *iPlanet UDS System Management Guide*

- *Programming with System Agents*

- *TOOL Reference Guide*

- *Using iPlanet UDS for OS/390*

Express Documentation

- *A Guide to Express*

- *Customizing Express Applications*

- *Express Installation Guide*

WebEnterprise and WebEnterprise Designer Documentation

- *A Guide to WebEnterprise*

- *Customizing WebEnterprise Designer Applications*

- *Getting Started with WebEnterprise Designer*

- *WebEnterprise Installation Guide*

Online Help

When you are using an iPlanet UDS development application, press the F1 key or use the Help menu to display online help. The help files are also available at the following location in your iPlanet UDS distribution:
`FORTE_ROOT/userapp/forte/cln/*.hlp`.

When you are using a script utility, such as Fscript or Escript, type help from the script shell for a description of all commands, or help `<command>` for help on a specific command.

iPlanet UDS Example Programs

A set of example programs is shipped with the iPlanet UDS product. The examples are located in subdirectories under `$FORTE_ROOT/install/examples`. The files containing the examples have a `.pex` suffix. You can search for TOOL commands or anything of special interest with operating system commands. The `.pex` files are text files, so it is safe to edit them, though you should only change private copies of the files.

Viewing and Searching PDF Files

You can view and search iPlanet UDS documentation PDF files directly from the documentation CD-ROM, store them locally on your computer, or store them on a server for multiuser network access.

NOTE	You need Acrobat Reader 4.0+ to view and print the files. Acrobat Reader with Search is recommended and is available as a free download from http://www.adobe.com. If you do not use Acrobat Reader with Search, you can only view and print files; you cannot search across the collection of files.

➤ **To copy the documentation to a client or server**

1. Copy the `doc` directory and its contents from the CD-ROM to the client or server hard disk.

    You can specify any convenient location for the `doc` directory; the location is not dependent on the iPlanet UDS distribution.

2. Set up a directory structure that keeps the `udsdoc.pdf` and the `uds` directory in the same relative location.

    The directory structure must be preserved to use the Acrobat search feature.

NOTE	To uninstall the documentation, delete the `doc` directory.

➤ **To view and search the documentation**

1. Open the file `udsdoc.pdf`, located in the `doc` directory.

2. Click the Search button at the bottom of the page or select Edit > Search > Query.

3. Enter the word or text string you are looking for in the Find Results Containing Text field of the Adobe Acrobat Search dialog box, and click Search.

    A Search Results window displays the documents that contain the desired text. If more than one document from the collection contains the desired text, they are ranked for relevancy.

NOTE	For details on how to expand or limit a search query using wild-card characters and operators, see the Adobe Acrobat Help.

4. Click the document title with the highest relevance (usually the first one in the list or with a solid-filled icon) to display the document.

    All occurrences of the word or phrase on a page are highlighted.

5. Click the buttons on the Acrobat Reader toolbar or use shortcut keys to navigate through the search results, as shown in the following table:

Toolbar Button	Keyboard Command
Next Highlight	Ctrl+]
Previous Highlight	Ctrl+[
Next Document	Ctrl+Shift+]

    To return to the `udsdoc.pdf` file, click the Homepage bookmark at the top of the bookmarks list.

6. To revisit the query results, click the Results button at the bottom of the `udsdoc.pdf` home page or select Edit > Search > Results.

Express Application Architecture

This chapter discusses the architecture of a generated Express application by examining the supplied and generated classes and illustrating how they interact.

Topics covered in this chapter include:

- runtime interaction between the client and business service

- a description of the supplied projects

- a description of the generated projects

- the relationship between generated classes and supplied classes

- the flow of control of between methods and objects during runtime

- the effect properties in the Business Model and Application Model Workshops have on generated classes

Throughout this chapter, examples of general topics are drawn from the Tutorial application. For example, you will see references to CUSTOMERORDERWindow and CUSTOMERORDERQuery—these classes exist in the generated TutorialAppWindows and TutorialServices projects.

Overview

The classes Express generates are used in one of the two runtime partitions, and occasionally both (for example, CUSTOMERORDERClass). In general, everything in the generated *business_model*Services (TutorialServices) project is used in the server partition, and everything in the generated *appl_model*Windows (TutorialAppWindows) project is used in the client partition. The following are the exceptions:

• Objects of class *business_class*Class and *business_class*Query are used in both partitions; they are defined in the *business_model*Services project.

• Class *business_model*Client is defined in project *business_model*Services, but objects of this class are used in the client partition.

Express services As described in *A Guide to Express*, you create the Express services that manage the business classes in your application. These Express services define the Service and DBService service objects for your application.

The Service service object maintains no state and all communication is message duration. Therefore, if a transaction is to span multiple messages, it must be managed from the client (Express handles this automatically if you select either the DB: Native Locking or DB: Explicit Locking options for the Concurrency property in the Business Model Workshop; see *A Guide to Express* for more information about the Concurrency property).

In addition, because communication on the Service service object is message duration, it can be load balanced.

The following diagram illustrates the two partitions that will exist at runtime in the deployed Tutorial application and shows key objects in each. Objects are listed in ellipses, and arrows between ellipses mean "invokes methods on." The prefix "a" means an instance of the class with that name (for example, aLINEITEMWindow). Multiple ellipses around an object name means that there may be many such objects (for example, an array of).

Note that you can customize every object in this diagram, except aDBSession, by modifying its class definition. See Chapter 2, "Customizing Express Applications," for details.

Figure 1-1   TutorialApp Partitions and Objects
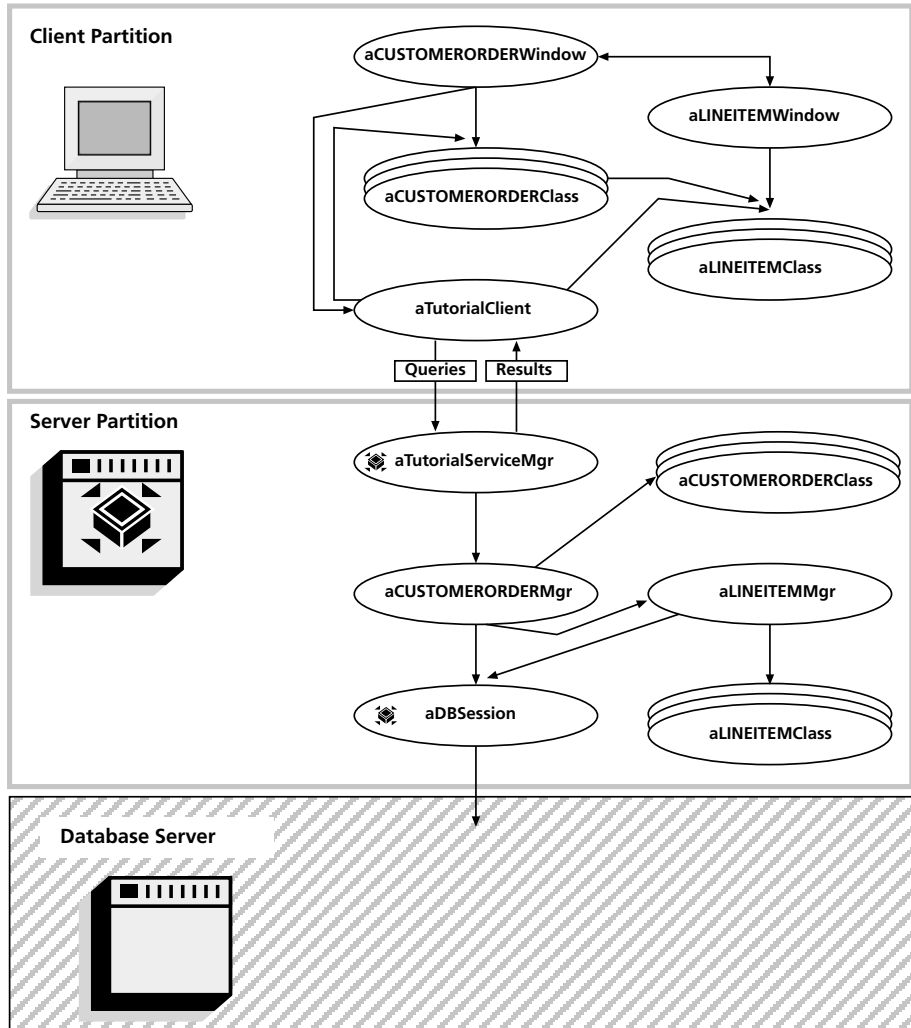
Express Projects

Every Express application has as supplier projects ExpressServices and ExpressWindows. The ExpressServices project contains a set of superclasses that provide server functionality for the classes generated from business models. The ExpressWindows project contains a set of superclasses that provide client functionality for the classes generated from application models. For full descriptions of the ExpressServices classes and the ExpressWindows classes, see the iPlanet UDS online Help.

The generated projects are discussed briefly in this section and in more detail in "Classes Generated from the Business Model" on page 26 and "Classes Generated from the Application Model" on page 28.

The class hierarchies of the supplied projects are shown in the figures below:

Figure 1-2   ExpressServices Class Hierarchy
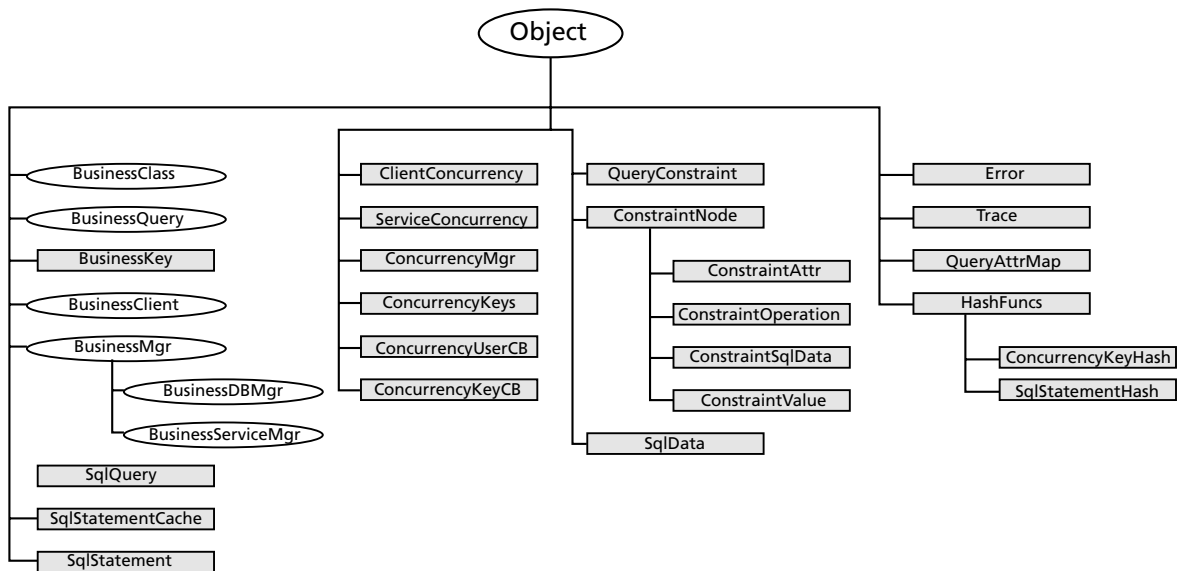
Figure 1-3   ExpressWindows Class Hierarchy
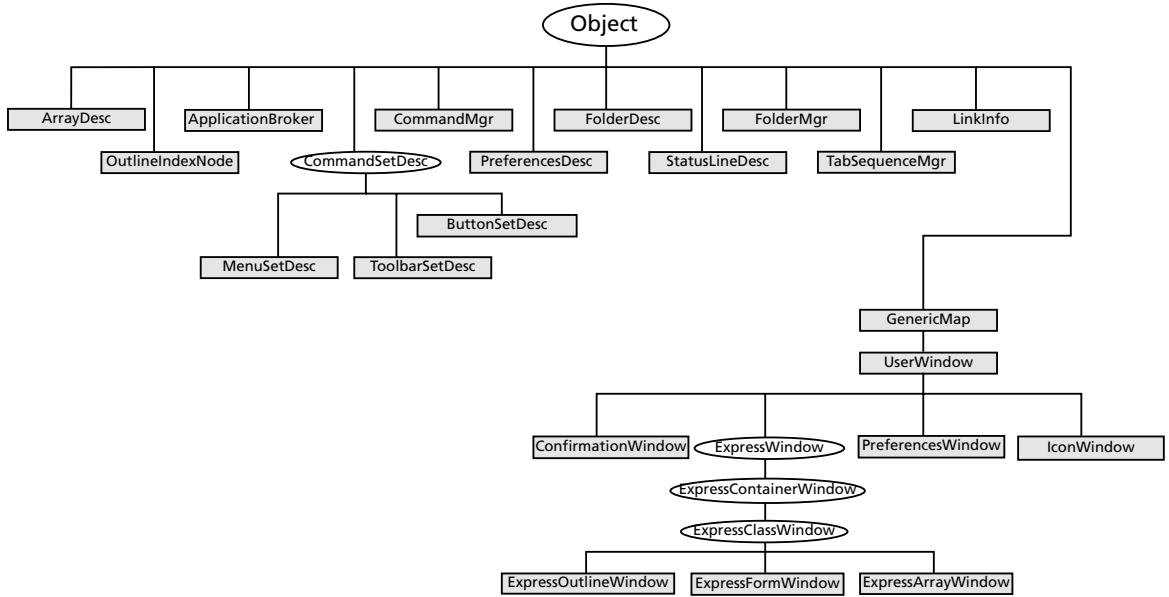
business_modelServices Project

Express generates the *business_model*Services project, which contains several classes based on the business model that support access to the underlying database. This project also contains the service object that manages the interaction with the database, including retrieving data and updating data with SQL statements. All the generated classes in *business_model*Services are ultimately subclasses of classes defined in the ExpressServices project. The *business_model*Services project includes the ExpressDomains project as a supplier plan.

Express uses the names of the business classes in the model and the name of the model itself to define the names of generated projects and classes.

The *business_model*Services project is a supplier project to the *appl_model*Windows project (described below) that Express generates from an application model that uses this business model as a supplier.

appl_modelWindows Project

Express generates the *appl_model*Windows project, which contains the window classes based on the application model. These window classes provide the graphical user interface to the application and define the flow between windows and the presentation of data in the windows. All the window classes generated

from the application model are subclasses of classes defined in the ExpressWindows project. The *appl_model*Windows project includes as supplier plans the *business_model*Services projects for the business models this application model uses. Other supplier plans include ExpressDomains, ExpressServices, and ExpressWindows.

Classes Generated from the Business Model

This section discusses the *business_model*Services classes generated from the business model. To generate these server classes, you can either use the Generate Server Code command in the Business Model Workshop or the `CompilePlan` command in the Fscript utility.

business_modelServices Project Classes

Express generates the following classes in the *business_model*Services project to store data and describe queries used for the business classes.

Generated Class	Superclass	Custom?	Description
*business_class*Class	BusinessClass (ExpressServices project)	No	Based on the business class defined in the business model. Each object stores a record of data from the table on which the business class is based.
*business_class*Query	BusinessQuery (ExpressServices project)	No	Represents the information used to retrieve the data for the *business_class*Class from the underlying database; also represents information to update the underlying database. This class defines constants, used as attribute indexes, for each of the attributes defined for the *business_class*Class.

Generated Class	Superclass	Custom?	Description
*service_name*BaseClient	BusinessClient (ExpressServices project)	No	Provides the interface between the application client and the *service_name*BaseMgr. Lets the client select and update *business_class*BaseClass objects, which are managed by the corresponding *service_name*ServiceMgr.
*service_name*Client	*service_name*Base Client	Yes	Contains your customizations for *service_name*BaseClient. This class is not regenerated when you regenerate your business model code, preserving your customizations.
*business_class*Mgr	BusinessDBMgr (ExpressServices project)	No	Manages select and update operations on the underlying database server.
*service_name*BaseServiceMgr	BusinessService Mgr	No	Accepts select and update requests from BusinessClient objects and sends them to the *business_class*Mgr.
*service_name*ServiceMgr	*service_name*Base ServiceMgr	Yes	Contains your customizations for *service_name*BaseServiceMgr.

If a class is Custom (indicated by a Yes in the Custom column), it is a customizable class and will not be regenerated when you regenerate the business model, preserving your customizations.

Express also generates the following service objects in the *business_model*Services project:

Generated Service Object	Class Defined By	Description
*service_name*DBService	DBSession (GenericDBMS project)	Provides the database session.
*service_name*Service	*service_name*ServiceMgr	Manages database interaction.

For information about using and partitioning these service objects, see *A Guide to Express*.

Customizable Subclasses in business_modelServices

By default, Express does not generate customizable subclasses of *business_class*Class, *business_class*Query, and *business_class*Mgr. For information about creating customizable subclasses of these *business_model*Services project classes, see "Creating Customizable Classes" on page 78.

Classes Generated from the Application Model

This section describes the classes in the generated *appl_model*Windows project. To generate these client classes from the application model, you can either use the Generate Client Code command under the File menu in the Application Model Workshop, or use the CompilePlan command in the Fscript utility.

appl_modelWindows Project Classes

Express generates the following classes in the *appl_model*Windows project to implement the user application defined in the application model.

Generated Class	Superclass	Custom?	Description
*appl_window*Window	ExpressArrayWindow, ExpressFormWindow, or ExpressOutlineWindow (ExpressWindows project)	No	Contains the information about the appearance and behavior of a window in an application. This class includes the code generated for the widgets that you defined for the window, as well as the attribute, methods, constants, events, and event handlers required to implement the functions and behavior that you defined for this window.
			*appl_window*Window can be a subclass of three different superclasses, depending on whether you have set the Layout of Fields Property in the Application Model Workshop to Form, Array, or Outline. See *A Guide to Express* for information about the Layout of Fields property.
*appl_window*Node	DisplayNode (Display library)	No	Displays data in an outline field. This class is generated only if you set the Layout of Fields property to Outline in the Application Model Workshop. See *A Guide to Express* for information about the Layout of Fields property.
*appl_model*BaseBroker	ApplicationBroker (ExpressWindows project)	No	Defines the TheBroker service object, which contains the generated preferences for the application and keeps track of all open windows in the application.
*appl_model*Broker	*appl_model*BaseBroker	Yes	Contains your customizations for *application_mode*BaseBroker. This class is not regenerated when you regenerate your application model code, preserving your customizations.

If a class is Custom (indicated by a Yes in the Custom column), it is a customizable class and will not be regenerated when you regenerate the application model, preserving your customizations.

Express also generates the following service object in the *appl_model*Windows project:

Generated Service Object	Class Defined By	Description
TheBroker	*appl_model*BaseBroker	Handles and maintains application preferences. Keeps a list of open windows (for Exit All).

For information about using and partitioning TheBroker, see *A Guide to Express*. For information about the generated classes and their attributes, methods, events, and event handlers, see the iPlanet UDS online Help.

Customizable Subclasses in appl_modelWindows

By default, Express does not generate customizable subclasses of *appl_window*Window and *appl_window*Node. For information about creating customizable subclasses of these *appl_model*Windows project classes, see "Creating Customizable Classes" on page 78.

Class Diagrams

The class diagrams in the following sections use the OMT notation described in the book "Object-Oriented Modeling and Design" to show Express classes and their relationships. The diagrams use a few extensions, influenced by those used in the book "Design Patterns":

• Arrows between classes mean that one class contains an attribute whose type is that of the class to which the arrow points. The name of the attribute is given by the text near the base of the arrow. A dot means an array of the other type is referenced. For example, in Figure 1-4, class SuperclassName contains an attribute named Refclass1, whose type is array of Otherclass1; SuperclassName also contains an attribute named Refclass2, whose type is Otherclass2.

• If one Express class references another, and the referenced class is described in another diagram or in another manual, then the bounding box of the referenced class will be a dotted line, and details about the class are usually not given. In Figure 1-4, OtherClass1 is described in another diagram.

- Each class displays a subset of the methods, attributes, event handlers, and constants used in the Tutorial application, or in Chapter 2, "Customizing Express Applications." These elements are indicated by the appearance of an icon —the same icon you see in the project workshop.

Figure 1-4   Example Class Diagram
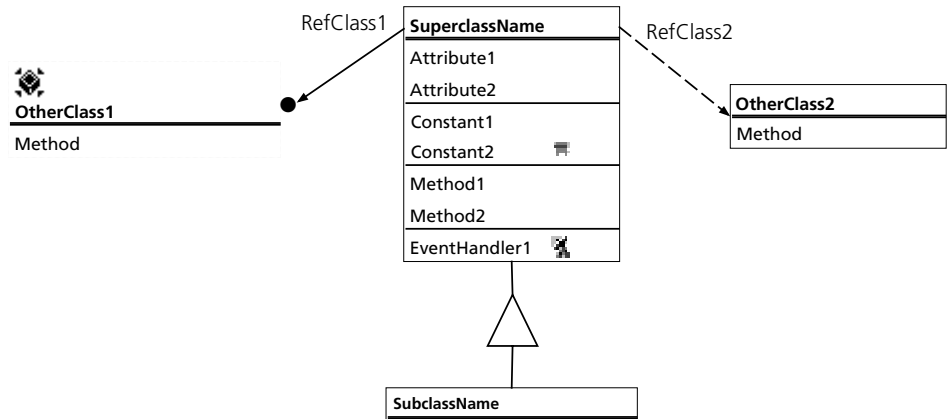
- Every class diagram will have at least two classes in an inheritance hierarchy and frequently three. The two-class hierarchy represents classes for which a customizable subclass is not generated by default. The three-class hierarchy represents those classes for which a customizable class is always generated. The OMT class diagrams for both types of classes are represented in Figure 1-5:

Figure 1-5   Class Inheritance Hierarchies
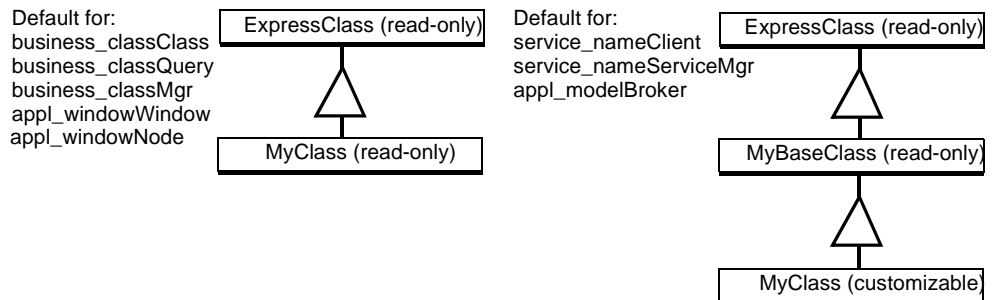
In all such class diagrams, the following is true:

- ExpressClass is a superclass in either ExpressWindows or ExpressServices. This superclass is not affected by code generation, but supports all types of classes that can be generated below it in the class hierarchy. In hierarchies that involve an ExpressWindow, ExpressClass will inherit from one of several classes from the ExpressWindows project. Each ExpressClass is described in the iPlanet UDS online Help.

- *my*Class (read-only) and *my*BaseClass (read-only) are generated classes containing specific characteristics that you specified in the application or business model. For example, in the Tutorial application, CUSTOMERORDERBaseWindow contains all the fields, mapped attributes, and methods to give the generated CUSTOMERORDERWindow the behaviors you specified in the Application Model Workshop.

- *my*Class (customizable) is generated the first time you generate only—it is initially empty. This class is where you will make your customizations. You are free to create methods and attributes on this class that will extend the behaviors found in its superclasses. Modifications you make to this class are safe because this class will not be affected by subsequent code generation. For information on creating customizable subclasses when they are not generated by default, see "Creating Customizable Classes" on page 78.

Declared Type and Runtime Type

If a class has an attribute that references a superclass in a hierarchy, then at runtime the type of the object referenced by that attribute will actually be the customizable subclass. For example, in Figure 1-7 on page 35, the BusinessQuery class has an attribute OriginalClass of type BusinessClass. At runtime, the attribute will be instantiated as a reference to the customizable CustomerOrderClass.

This customizable subclass reference gives you control over the runtime application behavior, because your subclass object and its customizations will be present wherever you see a reference to the base class in the class diagrams in this chapter.

Remember, though, to access attributes, methods, or events defined only for the runtime class, you must cast the object. Casting means identifying the class of a particular object. See the *TOOL Reference Guide* for more information about casting.

Classes Generated from the Tutorial Application

This section uses OMT class diagrams to describe the Express classes generated in the Tutorial application (see for information about how to read the class diagrams). Every class description includes a diagram showing the class's inheritance and relationships to other classes. Each diagram also displays commonly used methods, attributes, event handlers, and constants defined in the class.

Not all classes in the Tutorial application are shown in the following diagrams, because many of them are similar. For example, the class diagram for CUSTOMERORDERWindow is quite similar to that for LINEITEMWindow, so only CUSTOMERORDERWindow is shown. In general, the diagrams that follow focus on the CUSTOMERORDER business class.

Also note that no superclasses outside of Express projects are shown in the class hierarchies. For example, Object is the root class for every superclass in every diagram, but it is not shown.

Express generates the following classes and service objects for the Tutorial business model:

Figure 1-6   TutorialServices Project
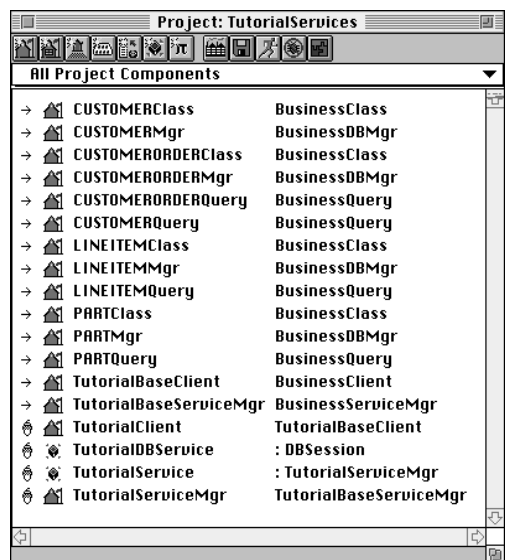
Notice that there is a set of classes for each class in the business model: Customer, CustomerOrder, LineItem, and Part. This section will describe only the classes based on the CustomerOrder business class. Information about CustomerOrder applies to the other business classes as well.

NOTE	The Tutorial application has no server-side customizations. If there were server-side customizations of, for example, the CUSTOMERORDERClass, Figure 1-6 would show CUSTOMERORDERClass to have a superclass named CUSTOMERORDERBaseClass, which would be listed as a subclass of BusinessClass.

The subset of classes generated based on the CustomerOrder business class includes:

Generic Class	CustomerOrderClass
*business_class*Class	CustomerOrderClass
*business_class*Query	CustomerOrderQuery
*business_class*Mgr	CustomerOrderMgr
*business_model*Client	TutorialClient (for each business model)
*business_model*ServiceMgr	TutorialServiceMgr (for each Express service)
*service_name*Service	TutorialService (one for each Express service)
*service_name*DBService	TutorialDBService (one for each Express service)
eClient	TutorialClient (one for each Express service)

CUSTOMERORDERClass and CUSTOMERORDERQuery

CustomerOrderClass represents the data in the CustomerOrder table. Any time you want to retrieve or modify data in this table, you will use a CustomerOrderQuery object. Queries, which are objects of type *business_class*Query (CUSTOMERORDERQuery), are passed from the client to the business service. For Select queries, the business service will return a result set consisting of an array of objects of type *business_class*Class (CUSTOMERORDERClass).

Figure 1-7   CustomerOrderClass and CustomerOrderQuery
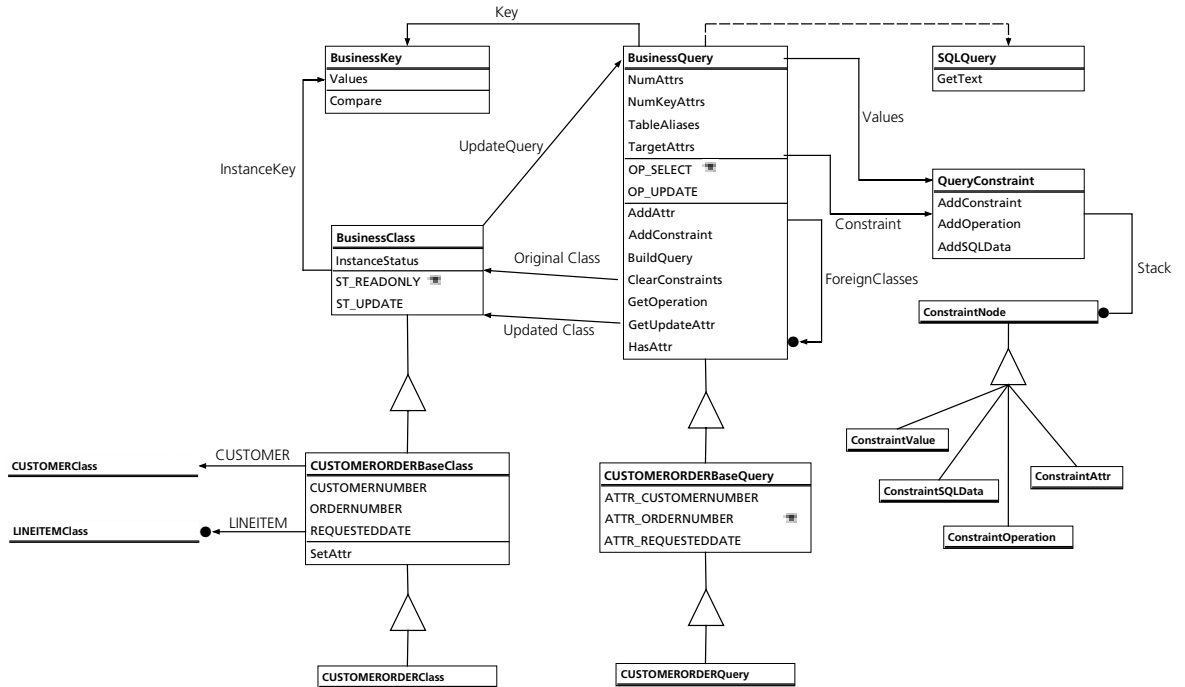
BusinessClass

BusinessClass is the superclass of CustomerOrderBaseClass. The following are commonly used attributes found on BusinessClass:

Attribute	Description
InstanceStatus	Provides the update status of each CustomerOrderClass object. Potential status values are derived from properties set for the class in the Application Model Workshop.
UpdateQuery	Points to the CustomerOrderQuery object associated with the CustomerOrderClass object. The query may be an Update, Insert, or Delete query, depending on the application and the actions taken by the user.
InstanceKey	References a BusinessKey object that tracks the originally selected values for the primary key fields in the current CustomerOrderClass object. The key values are used in the WHERE clause for Update and Delete queries.

BusinessClass is described in detail in the iPlanet UDS online Help.

CUSTOMERORDER[Base]Class

CustomerOrderClass is the default name for the read-only, generated class that contains information specific to the CustomerOrder business class. When a customization is created for this class, it is automatically renamed to "CustomerOrderBaseClass," and a new, customizable subclass is created and named "CustomerOrderClass." There is an attribute in CustomerOrderBaseClass for every attribute specified for the CustomerOrder class in the Business Model Workshop. There are additional attributes found on this class that Express creates based on associations you specified in the Business Model Workshop.

The table below describes important attributes of the CustomerOrder[Base]Class:

Attribute	Description
CustomerNumber OrderNumber RequestedDate	Correspond to attributes defined on the business class in the Business Model Workshop. See *A Guide to Express* for information about attributes.
Customer	Contains a reference to a CustomerClass object, based on the many-to-one association between CustomerOrder and Customer in the Tutorial business model.
LineItem	Contains a reference to an array of LineItemClass objects, based on the one-to-many association between CustomerOrder and LineItem in the Tutorial business model.

CUSTOMERORDERClass

CustomerOrderClass is the customizable, generated subclass of CustomerOrderBaseClass. This class is initially empty, but you can make modifications to CustomerOrderBaseClass behavior here by adding attributes and overriding methods defined in its superclasses.

BusinessQuery

BusinessQuery is the superclass of CustomerOrder[Base]Query and provides the methods and attributes necessary for creating queries based on data from a business class. The text of a query is created by a SqlQuery object; a SqlQuery object is created by a BusinessQuery object. See the iPlanet UDS online Help.

The following table describes common attributes on the BusinessQuery class:

Attribute	Description
OriginalClass	Points to the original CustomerOrder object, before the user has updated it. Express uses this attribute when the Concurrency property is set to Optimistic:Verify. Otherwise, this attribute is NIL. See *A Guide to Express* for information about the Concurrency property.
UpdatedClass	Points to the updated CustomerOrder object. Express uses this attribute when the Send Only Changes on Update property is off. Otherwise, this attribute is NIL. See *A Guide to Express* for information about the Send Only Changes on Update property.
InstanceKey	Points to the original primary key values. These values are necessary to execute an Update or Delete query.
Values	Points to the new values for Update queries. See the iPlanet UDS online Help.
Constraint	Points to the WHERE clause constraints. See the iPlanet UDS online Help.

BusinessQuery is described in detail in the iPlanet UDS online Help.

CUSTOMERORDER[Base]Query

CustomerOrderQuery is the default name for the read-only, generated class that contains information specific to the CustomerOrderQuery business class. When a customization is created for this class, it is automatically renamed to "CustomerOrderBaseQuery," and a new, customizable subclass is created and named "CustomerOrderQuery." The integer "ATTR_..." constants defined in this class are used in many methods to identify the attributes on the CustomerOrder business class. See Chapter 2, "Customizing Express Applications," for more information and example uses.

CUSTOMERORDERQuery

CustomerOrderQuery is the customizable, generated class. This class is initially empty, but you can make modifications to the text of the query here by adding attributes and overriding methods defined in superclasses.

BusinessKey

The BusinessKey class defines primary key values. For more information, see the iPlanet UDS online Help.

SqlQuery

The SqlQuery class creates SQL text based on the data described by a BusinessQuery object. You can create a subclass of SqlQuery to modify the type of SQL Express generates. You will also have to override the GetSQLQuery method on the BusinessQuery class so it will create your SqlQuery subclass. For more information, see the iPlanet UDS online Help.

QueryConstraint

The QueryConstraint class contains the constraints for the BusinessQuery object. These constraints are a superset of the normal WHERE clause constraints; they may also include ORDER BY settings, and a limit on the number of selected rows.

The Stack attribute points to a list of constraints that is an array of ConstraintNode objects. References to a ConstraintNode will actually be references to a ConstraintNode subclass.

For more information, see the iPlanet UDS online Help.

TutorialClient Class

Express creates one BusinessClient object for each *business_model*ServiceMgr (see "CUSTOMERORDERMgr and TutorialServiceMgr" on page 41). Windows do not communicate with the business service directly, but use *business_model*Client objects (TutorialClient) to invoke methods in the business service.

TutorialClient is the subclass of BusinessClient that corresponds to TutorialServiceMgr. The hierarchy of the TutorialClient class is shown below.

Figure 1-8   TutorialClient
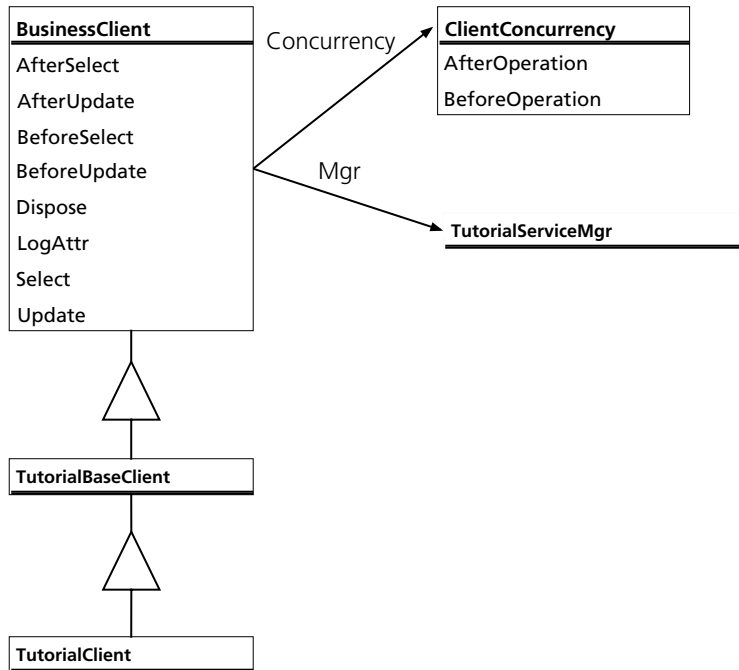
BusinessClient

BusinessClient is the superclass of TutorialBaseClient and provides methods and attributes that are responsible for all communication with the *business_model*Service service object on the server. There will be one BusinessClient object in the client partition for each *business_model*Service service object in the server partition.

Important attributes on the BusinessClient class are described below:

Attribute	Description
Concurrency	Contains a reference to a ClientConcurrency object, which controls database concurrency from the client side. See "ClientConcurrency" on page 40.
Mgr	Contains a reference to the TutorialServiceMgr.

Important methods on the BusinessClient class are described below:

Method	Description
LogAttr	Stores the new value for each CustomerOrderClass attribute for use in subsequent update or insert queries. The values are stored in the Values attributes of the CustomerOrderQuery object.

The BusinessClient class is described in detail in the iPlanet UDS online Help.

TutorialBaseClient

TutorialBaseClient is a read-only, generated class. This class provides no additional methods or attributes.

TutorialClient

TutorialClient is the customizable, generated subclass of TutorialBaseClient. This class is initially empty, but you can make modifications to TutorialBaseClient behavior here by adding attributes and overriding methods defined on the BusinessClient class.

ClientConcurrency

The ClientConcurrency class controls concurrency from the client. You specify concurrency options in the Business Model Workshop (see *A Guide to Express*). If you choose the Database: Explicit Locking or Database: Native Locking concurrency options, then the ClientConcurrency object will begin and end transactions. If you choose the Optimistic: Verify option, then each client message will be a separate transaction, beginning and ending on the server.

The ClientConcurrency class is primarily used internally and should not be customized.

CUSTOMERORDERMgr and TutorialServiceMgr

In the Tutorial application, there is one *business_model*Service service object, TutorialService, which is based on the Tutorial service (the TutorialService service object is of type TutorialServiceMgr). The Tutorial service was created by default when you created the Tutorial business model; you modify service properties, or create new services, in the Business Model Workshop. See *A Guide to Express* for more information about Express services.

For each business class, there is a corresponding class that manages communication between the business class and the database. For example, in the Tutorial application, the CustomerOrderClass has a corresponding CustomerOrderMgr object. The TutorialClient object passes queries to the TutorialService service object, which then communicates with the appropriate *business_class*Mgr class (in the case, CustomerOrderMgr). CustomerOrderMgr handles communication with the database.

Figure 1-9   CustomerOrderMgr and TutorialServiceMgr
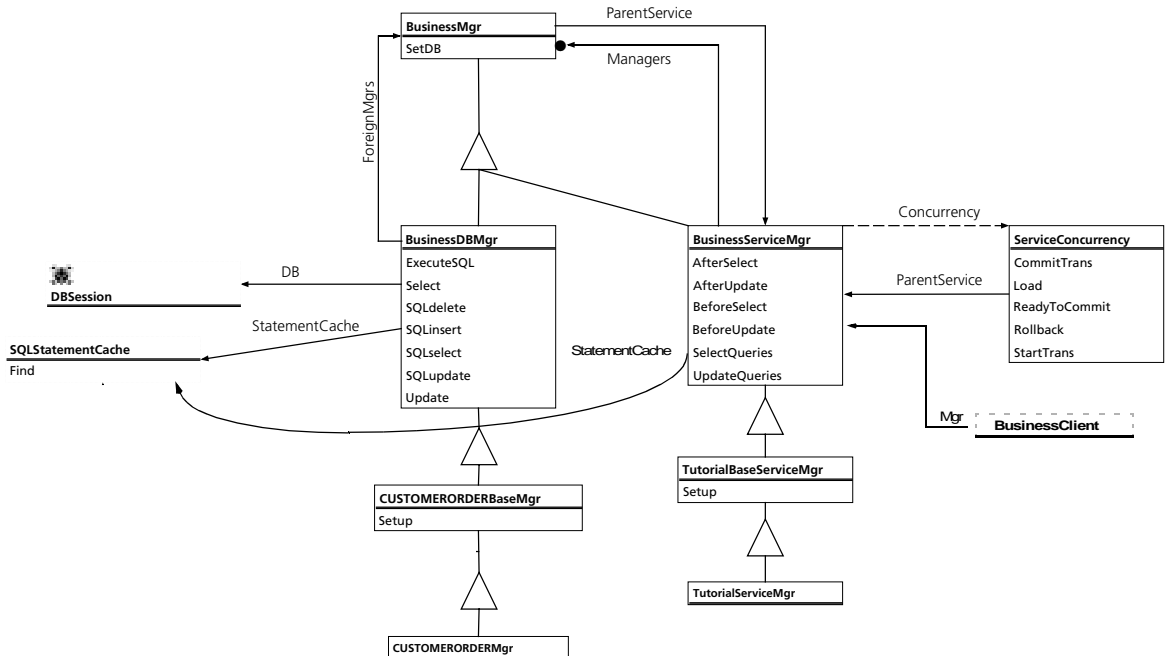
BusinessMgr

BusinessMgr is the superclass of BusinessDBMgr. BusinessMgr objects create business classes, retrieve data for them, and perform updates to track changes in business classes. For example, the ParentService attribute contains a reference to the service object that passes queries to CustomerOrderMgr. See the iPlanet UDS online Help for more information.

BusinessDBMgr

BusinessDBMgr is the superclass of CustomerOrderBaseMgr. This class provides methods for executing SQL queries based on any business class and attributes that handle information about the execution of queries. Important attributes and methods are described below:

Attribute	Description
DB	Contains a reference to the database session object to use for all queries.
StatementCache	Contains a reference to the cache that will contain previously run queries. The cache contains the results of the Prepare and DescribeTable methods on DBSession. Express first checks each query against the cache to see if an identical query has already run on the current session. If there is an identical query, Express uses the cached results from the Prepare method.
ForeignMgrs	Contains a reference to the other BusinessDBMgr objects with which the BusinessDBMgr object may need to communicate based on associations in the Business Model Workshop. This attribute is used only for Select queries and one-to-many relationships.

Method	Description
SetDB	Sets which database session to use (in this example, TutorialDBService).

The BusinessDBMgr class is described in the iPlanet UDS online Help.

CUSTOMERORDER[Base]Mgr

CustomerOrderMgr is the default name for the read-only, generated class that contains information specific to the CustomerOrderMgr business class. When a customization is created for this class, it is automatically renamed to "CustomerOrderBaseMgr," and a new, customizable subclass is created and named "CustomerOrderMgr." The Setup method on this class populates the BusinessDBMgr.ForeignMgrs array.

CUSTOMERORDERMgr

The CustomerOrderMgr class is a customizable, generated class. This class is initially empty, but you can make modifications to CustomerOrderBaseMgr behavior here by adding attributes and overriding methods defined in the superclasses. The business service will contain one object of this class for each business class managed by the server—for example, there will be a CustomerOrderMgr, a LineItemMgr, and so on. The Service service object (TutorialService) then hands queries it receives from the client (from TutorialClient) to one of the appropriate *business_class*Mgr objects, according to the business class of the query.

BusinessServiceMgr

BusinessServiceMgr is the superclass of TutorialBaseServiceMgr. This class accepts select and update requests from BusinessClient class objects (TutorialClient) and sends them to the appropriate subclass of BusinessMgr (in this case, CustomerOrderBaseMgr).

Important attributes on the BusinesServiceMgr class are described in the table below:

Attribute	Description
Managers	Contains a reference to an array of appropriate *business_class*Mgr objects.
StatementCache	Contains a reference to the cache that will contain previously run queries. The cache contains the results of the Prepare and DescribeTable methods on DBSession. Express first checks each query against the cache to see if an identical query has already run on the current session. If there is an identical query, Express uses the cached Prepare method results.
Concurrency	Contains a reference to a ServiceConcurrency object instantiated by the BusinessServiceMgr.

Important methods on the BusinesServiceMgr class are described in the table below:

Method	Description
AfterSelect	Empty method intended for you to override in TutorialServiceMgr. Enter code in this method to perform an operation after a select.
AfterUpdate	Empty method intended for you to override in TutorialServiceMgr. Enter code in this method to perform an operation after an update.
BeforeSelect	Empty method intended for you to override in TutorialServiceMgr. Enter code in this method to perform an operation before a select.
BeforeUpdate	Empty method intended for you to override in TutorialServiceMgr. Enter code in this method to perform an operation before an update.
SelectQueries pdateQueries	Walks through the queries received from the client and asks the appropriate *business_class*Mgr to execute each query. Note that because these methods affect queries, you will usually override methods defined in BusinessDBMgr, for each *business_class*Class, because that will only affect queries for that class. These methods are invoked when the client wants to pass queries to the business server. See "Press Search Button" on page 58 and "Press Save Button" on page 59.

The BusinessServiceMgr class is described in the iPlanet UDS online Help.

TutorialBaseServiceMgr

TutorialBaseServiceMgr is a read-only generated class. The Setup method in this class determines to which *business_class*Mgr objects to pass queries.

TutorialServiceMgr

TutorialServiceMgr is a customizable, generated class. This class is initially empty, but you can make modifications to TutorialBaseServiceMgr behavior here by adding attributes and overriding methods defined in superclasses. Express creates a service object based on this class. You determine the number of service objects for an application in the Business Model Workshop (see *A Guide to Express*).

ServiceConcurrency

The ServiceConcurrency class is used by the BusinessServiceMgr to control concurrency during query execution. Note that this class uses the concurrency specifications set in the Business Model Workshop (see *A Guide to Express*). This class may change the query, or execute other queries to support the specified concurrency mode.

The ServiceConcurrency class is primarily used internally and should not be customized.

TutorialAppWindows Project

Express generates the following classes and service objects for the TutorialApp application model:

Figure 1-10   TutorialAppWindows Project
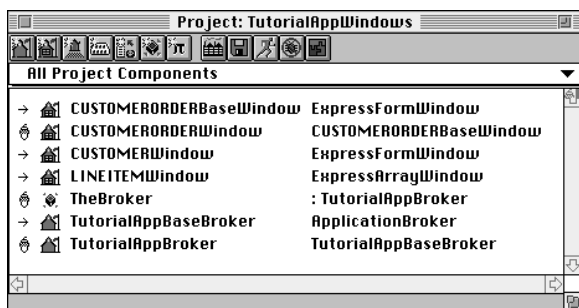
Notice that there is a class for each window class in the application model: Customer, CustomerOrder, LineItem, and Part. There is only one customized class in the Tutorial application: the CustomerOrder class. Therefore, Figure 1-10 shows only a Base class/customizable class set only for this class. If you had customized the other window classes, or if you had turned on the Always Generate Custom Classes toggle, Figure 1-10 would show a duo set for the other window classes. This section describes only the classes based on the CustomerOrder business class. Information about CustomerOrder applies to the other window classes as well.

For information on the Always Generate Custom Classes toggle, see "Custom Generation Options" on page 70.

The classes from the TutorialAppWindows project for the CustomerOrderWindow class include:

Generic Class	CustomerOrderWindowClass
*business_class*Window	CustomerOrderWindow
*business_class*BaseWindow	CustomerOrderBaseWindow
*business_model*BaseBroker	TutorialAppBaseBroker (for each application model)
*business_model*Broker	TutorialAppBroker (for each application model)
TheBroker	TutorialAppBroker (for each application model)

CUSTOMERORDERWindow Class

The Express generates the following classes for the CustomerOrderWindow class. All the supplied (non-generated) classes in this section are described in the iPlanet UDS online Help.

Figure 1-11   CustomerOrderWindow Class
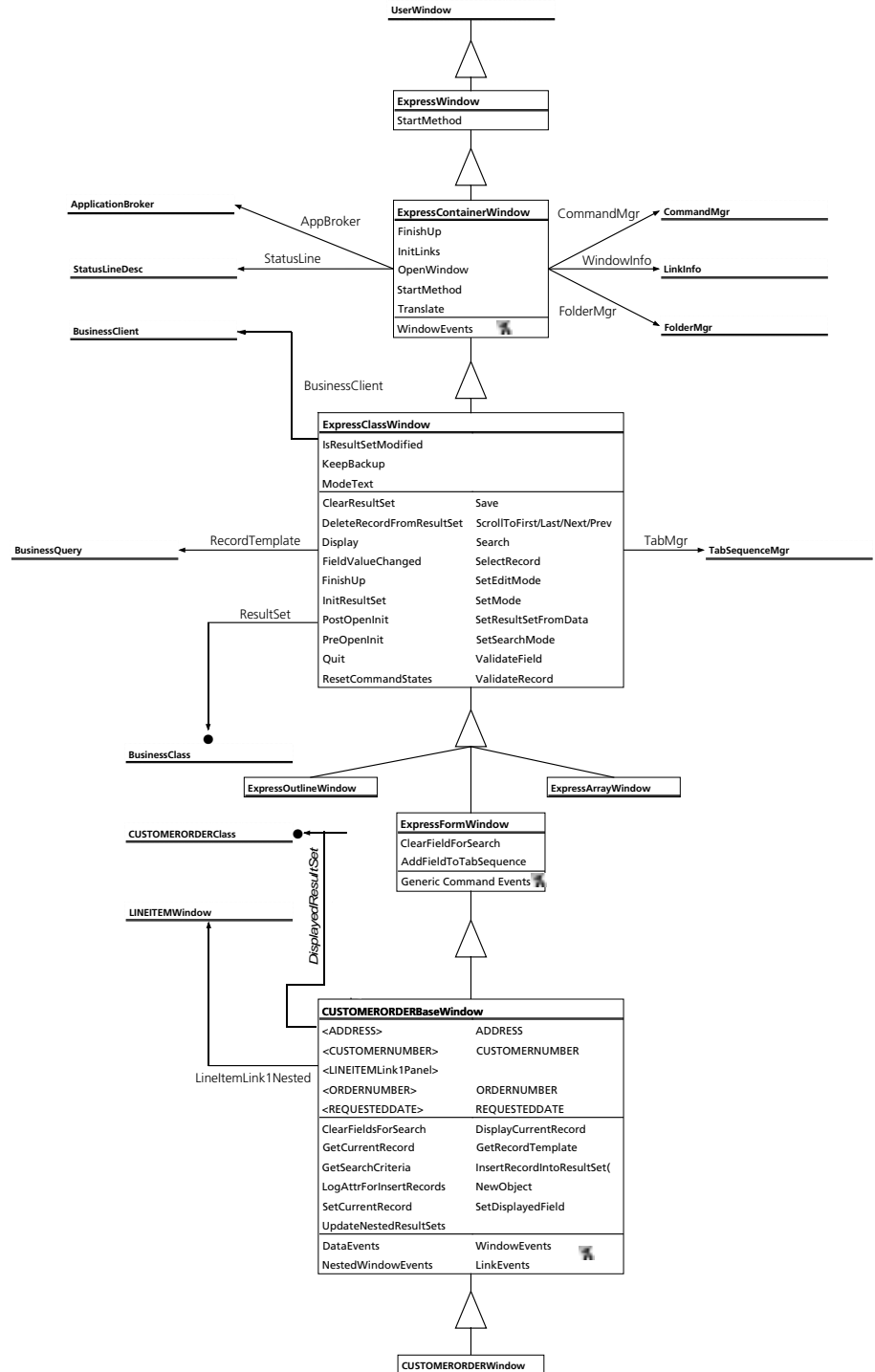
ExpressWindow

ExpressWindow is the superclass for all window classes. This class defines the generic entry point (StartMethod) used for all windows. For more information, see the iPlanet UDS online Help.

ExpressContainerWindow

The ExpressContainerWindow class provides support for links to other windows. You specify the types of links between windows in the Application Model Workshop (see *A Guide to Express* for information about links).

Important attributes on the ExpressContainerWindow class are described in the table below:

Attribute	Description
AppBroker	Contains a reference to the Application Broker service object (TheBroker).
WindowInfo	Contains a reference to the LinkInfo object that describes the type of command set specified for the window. See "LinkInfo" on page 55.

For more information, see the iPlanet UDS online Help.

ExpressClassWindow

The ExpressClassWindow class provides support for windows that display data associated with another business class.

Attribute	Description
BusinessClient	Contains a reference to the object (TutorialClient) that handles all communication between client and business server.
IsResultSetModified	Is set to TRUE when the user changes the value of a field.
RecordTemplate	Contains a reference to the fields needed for the window, the maximum number of records to retrieve, and the sort order. The GetRecordTemplate method (on CustomerOrderBaseWindow) instantiates and initializes the BusinessQuery object referenced by RecordTemplate.
ResultSet	An array that contains a reference to data the window will display, including deleted rows.

Attribute	Description
WindowMode	Contains the current mode of the window (WM_EDIT or WM_SEARCH).

For more information, see the iPlanet UDS online Help.

ExpressFormWindow, ExpressArrayWindow, ExpressOutlineWindow

ExpressFormWindow, ExpressArrayWindow, and ExpressOutlineWindow provide support for the specific field layout of the generated window. CustomerOrderBaseWindow is a subclass of one of these classes—the specific class is determined by settings specified in the Application Model Workshop (see *A Guide to Express* for information about specifying window layout). For more information about the window superclasses, see the iPlanet UDS online Help.

CUSTOMERORDER[Base]Window

CustomerOrderWindow is the default name for the read-only, generated class that contains information specific to the CustomerOrderWindow business class. When a customization is created for this class, it is automatically renamed to "CustomerOrderBaseWindow," and a new, customizable subclass is created and named "CustomerOrderWindow." In the tutorial, you specified the CustomerOrder window to have a Form layout. As a result, CustomerOrder[Base]Window is a subclass of ExpressFormWindow.

CustomerOrder[Base]Window contains attributes for each displayed field and methods that manage the links out of the window. In addition, CustomerOrder[Base]Window contains the following useful attributes:

Attribute	Description
DisplayedResultSet	An array that contains a reference to the data the window will display (does not include a reference to deleted rows). If the Layout of Fields property is set to Array, then DisplayedResultSet is a mapped attribute for the array field. See *A Guide to Express* for information about the Layout of Fields property. For more information, see the iPlanet UDS online Help.
LineItemLink1Nested	Contains a reference to a nested window. This attribute is generated only if the window has a link to a nested window; the name of the attribute is based on the name of the link to the nested window.

CustomerOrder[Base]Window has several event handlers that manage events on generated fields and links to other windows.

Event Handler	Description
DataEvents	Handles events on the fields generated onto the window. For example, an AfterValueChange event is available for every field that has a Read status of Read/Write. See *A Guide to Express* for information about Read status.
LinkEvents	Is present if you define links to other windows in the Application Model Workshop.
NestedwindowEvents	Is generated in case the window is ever nested inside another window. This handler allows the nested window to register for events in the parent window's event loop.
WindowEvents	Contains the register statements to register the other event handlers.

For more information about the generated windows, see the iPlanet UDS online Help.

CUSTOMERORDERWindow

CustomerOrderWindow is a customizable, generated class. This class is initially empty, but you can make window and window class modifications here. See Chapter 2, "Customizing Express Applications," for details.

TutorialAppBroker Class

The TutorialAppBroker class is a subclass of the ApplicationBroker class. A service object name TheBroker is generated of type TutorialAppBroker.

Figure 1-12   TutorialAppBroker Class
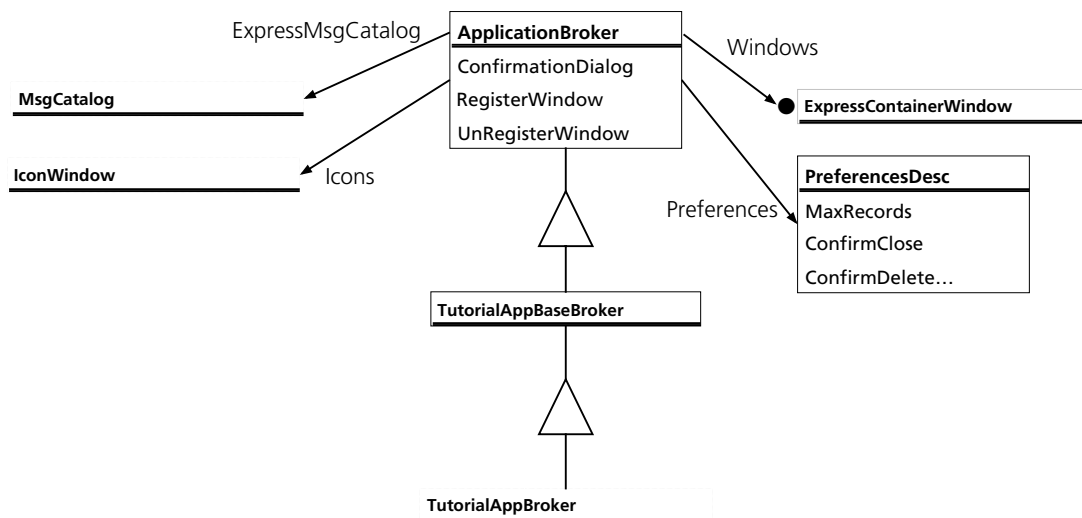
ApplicationBroker

ApplicationBroker is the superclass for TutorialAppBaseBroker and provides application-wide information, such as the current message catalog and preferences information. This class also controls application shutdown.

Useful attributes on the ApplicationBroker class are described below:

Attribute	Description
Windows	Contains an array that points to the open windows in the application.
Preferences	Contains a reference to the object that contains the preferences set for the application in the Application Model Workshop. See *A Guide to Express* for information about setting generated window preferences.

For more information, see the iPlanet UDS online Help.

TutorialAppBaseBroker

TutorialAppBaseBroker is a read-only, generated class containing the actual values of the generated application preferences specified in the application model. For more information, see the iPlanet UDS online Help.

TutorialAppBroker

TutorialAppBroker is a customizable, generated class. This class is initially empty, but you can make modifications to Application Broker behavior here by adding attributes and overriding methods defined in the superclasses.

TheBroker

A service object named TheBroker is created of type TutorialAppBroker. One TheBroker service object is created for each client project (in this case, TutorialAppWindows). TheBroker keeps track of all the open windows in the application—Express uses this information if the user selects the Exit All command from the File menu. TheBroker also manages the runtime preferences that were set in the Application Model Workshop (see *A Guide to Express*).

CommandMgr

The classes in this diagram are not designed to be customized in the generated Express application. They are shown here for completeness, because CommandMgr is referenced by the ExpressWindow classes, and to explain how the window commands are implemented. These classes are described in the iPlanet UDS online Help.

Figure 1-13   CommandMgr
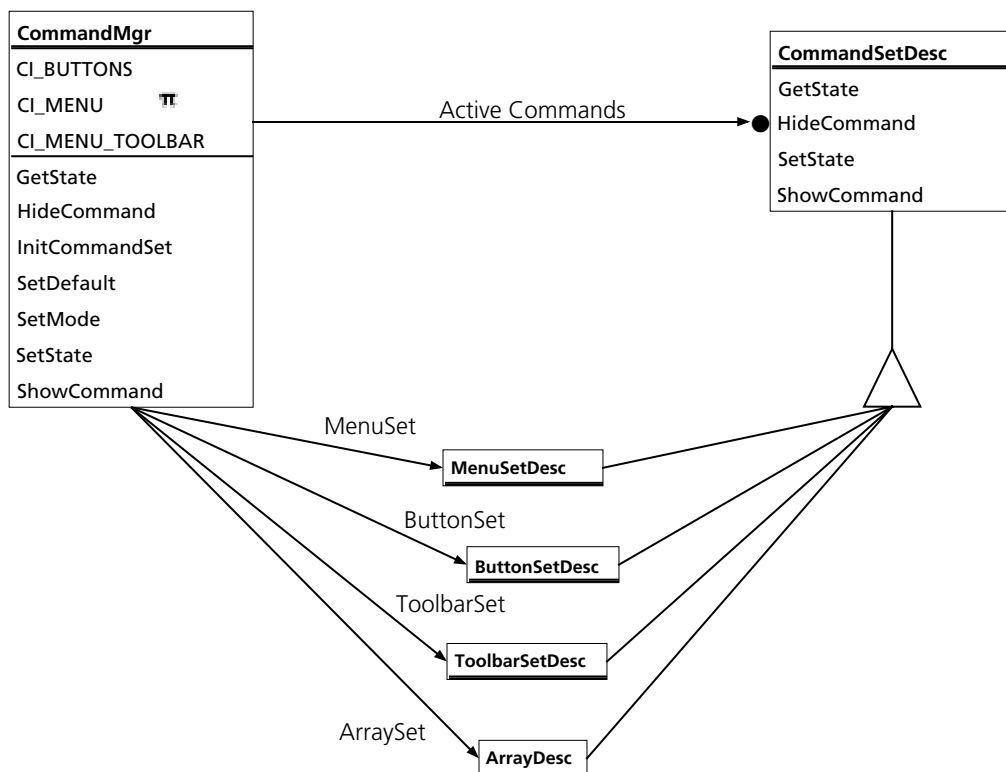
CommandMgr

The CommandMgr class handles the display of window buttons and menu commands, and also sets the window state. If you set the Adaptive Command Interface option on a window in the Application Model Workshop, then the CommandMgr object removes or adds commands and changes window mode, depending on what is appropriate.

For example, if the Adaptive Command Interface property is off, then Express uses the information set in the Command Set and Default Interface properties to determine which command widgets are generated onto a window. In addition, the LinkInfo object passed to the window will specify that the default command interface and command set be used.

If the Adaptive Command Interface property is on, then the values for the Command Set and Default Interface properties are ignored, and the window is generated as if the All Command Set was specified. Subsequently, when the window is invoked, Express uses the CommandSet and CommandInterface

attributes on the LinkInfo class to determine which menus, toolbars, and buttons are required on the window. Any unnecessary commands are removed. See *A Guide to Express* for more information about using the Adaptive Command Interface property.

Note, however, that if the window is the start window, then no LinkInfo object is passed to the window, and the Command Set and Default Interface properties for the window will be used to determine which commands and interfaces to display on the window.

ActiveCommands attribute   The ActiveCommands attribute on the CommandMgr class is an array that contains a reference to all the active command sets for the parent window. These command sets are referenced separately by the ButtonSet, MenuSet, and ToolbarSet attributes.

CommandSetDesc

CommandSetDesc is the superclass for all the classes that control command sets on a generated window. This class provides common functionality, such as hiding commands and changing command states, for all command sets.

ButtonSetDesc, MenuSetDesc, ToolBarSetDesc

The ButtonSetDesc, MenuSetDesc, and ToolbarSetDesc classes manage the behavior of each specific command set—Buttons, Menu and Toolbar. Methods in these classes override the common functionality provided by the CommandSetDesc class. There is one attribute on each class for every command provided on the interface. For example, ButtonSetDesc.DeleteCMD represents the delete button on a button interface.

LinkInfo

The LinkInfo object contains the information passed into a window when the window is called by a link. Express uses the LinkInfo object to pass information to a window and return information from a window. The LinkInfo object describes the caller, the query to run, and which broker and business class to use.

Figure 1-14   LinkInfo
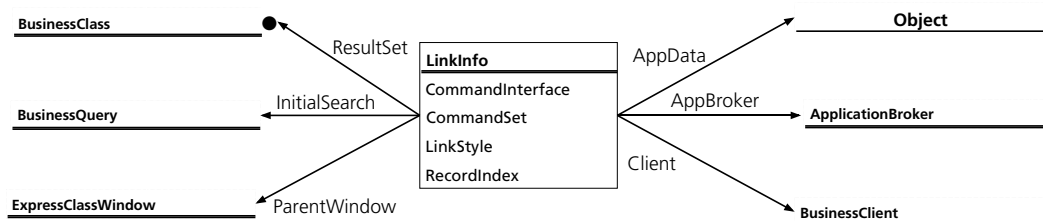
The following are useful attributes on the LinkInfo class.

Attribute	Description
AppBroker	Contains a reference to the ApplicationBroker object that the called window should use.
AppData	Contains a generic object pointer to pass application data to the called window.
Client	Contains a reference to the BusinessClient that the called window should use for any queries it needs to run.
InitialSearch	Contains the criteria for the initial search the called window will execute upon opening.
LinkStyle	Contains the link type, which is determined by the Link setting in the Link Properties dialog.
ParentWindow	Contains a reference to the calling window.
RecordIndex	Contains the record number in the result set to return to the caller (lookup links expect a BusinessClass object to be returned).
ResultSet	Contains a reference to the BusinessClass object that contains the original data passed to the called window. ResultSet is not used for output—any returned data is passed using the return code from the called window's Display method.

Runtime Scenarios

This section uses Object Interaction Diagrams to trace the interaction between objects during several key runtime scenarios in a generated Tutorial window (CUSTOMERORDERWindow with nested LINEITEMWindow). These diagrams show the objects involved in the processing and the methods they invoke on each other.

The diagrams show only the key objects involved in a scenario and the methods they invoke on each other. You can build diagrams like these on your own by running your application and stepping through it in the TOOL debugger.

Object Interaction Diagram Notation

The Interaction Diagrams are based on the Jacobson notation described in the book *Object-Oriented Software Engineering*. These diagrams illustrate runtime scenarios and the flow of control when Express applications perform particular actions.

Vertical lines in these diagrams represent objects. The object name is given at the top of the vertical line—its name is identical with its class name, but prefixed with the letter "a".

Objects invoke methods on each other, and these methods are represented by horizontal lines between objects with an arrow indicating the direction. The name of the method invoked is shown on the line. Thus, in Figure 1-15, object aClass1 invokes method MethodName() on class aClass2 with parameters 'params'.

Time flows from top to bottom. Thus, after aClass1 invokes aClass2.MethodName(), aClass2 then invokes aClass3.MethodName2().

If an object invokes a method on itself, this is shown as a method line that points back to the same object.

Figure 1-15   Example Object Interaction Diagram

aClass1   aClass2   aClass3

MethodName(args)

MethodName2(params)

MethodName3()

MethodName4()

Press Search Button

The diagram includes the following lifelines (object headers):

- aCUSTOMERORDERWindow
- aTutorialClient
- aTutorialServiceMgr
- aClientConcurrency
- aServiceConcurrency
- aCUSTOMERORDERMgr
- aCUSTOMERORDERQuery
- aSQLQuery
- aSQLStatementCache
- aDBSession
- aLINEITEMMgr

Messages shown in the sequence:

- GetSearchCriteria()
- Search(BusinessQuery)
- Select(BusinessQuery)
- BeforeOperation(transactionMode)
- BeforeSelect(BusinessQuery)
- SelectQueries(ClientID, array of BusinessQuery)
- Begin Dependent Transaction StartTrans()
- BeforeSelect(BusinessQuery)
- Select(CUSTOMERORDERQuery)
- BuildQuery(SQLQuery)
- SQLSelect(SQLQuery, SQLText)
- GetText()
- Find(SQLText)
- OpenCursor(SQLStatement)
- FetchCursor()
- CloseCursor()
- Select(LINEITEMQuery)
- AfterSelect()
- end transaction
- AfterSelect(Array of BusinessClass, BusinessQuery)
- AfterOperation(transactionMode)
- SetMode(WM_EDIT)
- SetResultSetFromData(data)

Note on LINEITEMQuery:

Perform same operations as above, but on LINEITEMQuery: BuildQuery(), GetText(), SQLSelect(), Find(), Open/Fetch/CloseCursor()

Press Save Button

Lifelines (top labels):
aCUSTOMERORDER Window · aTutorialClient · aClientConcurrency · aCUSTOMERORDERQuery · aTutorialServiceMgr · aServiceConcurrency · aCUSTOMERORDERBRMgr · aSQLQuery · aSQLStatementCache · aDBSession · aLINEITEMMgr · aCUSTOMERORDERClass

Messages:
- Save()
- ValidateRecord()
- LogattrForInsertRecords()
- UpdateNestedResultSets()
- Update(ResultSet)
- BeforeOperation(tranMode)
- BeforeUpdate(array of BusinessClass)
- AddAttr(Foreign Business Class)
- UpdateQueries(clientId, array of BusinessQuery)
- ReadyToCommit(clientId, array of BusinessQuery)
- Begin Dependent Transaction
- BeforeUpdate()
- Update(BusinessQuery)
- BuildQuery(SQLQuery)
- GetText()
- SQLInsert(), SQLUpdate(), or SQLDelete()
- ExecuteSQL()
- Find()
- Execute()
- Update(BusinessQuery)
- AfterUpdate()
- End Transaction
- CommitTrans(array of BusinessQuery, clientId, tranMode)
- Reset()
- AfterUpdate()
- AfterOperation()
- ResetCommandStates()

Window Startup

aCUSTOMERORDERWindow	aCommandMgr	aLINEITEMWindow	aCommandMgr	aTutorialAppBroker

Display()

PreOpenInit()

InitCommandSet()

InitLinks()

Translate()

PreOpenInit()

InitCommandSet

InitLinks()

Translate()

OpenWindow()

PostOpenInit()

InitResultSet()

SetResultSetFromData()

SetResultSetFromData()

RegisterWindowEvents()

RegisterNestedWindowEvents()

RegisterWindow()

Note that the Display method is not executed on the nested window. This is because the nested window's events will be registered as part of the parent window's event loop, so the nested window does not its own event loop.

Window Close With Unsaved Changes

aCUSTOMERORDERWindow	aTutorialClient	aCUSTOMERORDERClass	aTutorialAppBroker

HandleValueChange()

ValidateField()

LogAttr(BusinessClass, ATTR_...

SetAttr(ATTR_,DataValue)

FieldValueChanged(ATTR_, DataValue)

FinishUp()

ConfirmationDialog()

post AfterFinishUp
UnregisterWindow(CUSTOMERORDERWindow)

Dispose()

Workshop Properties and Generated Classes

This section discusses the effects of Business Model and Application Model property settings on the generated classes.

Business Model Workshop

The following sections describe the properties set in the Business Model Workshop and their effect on the classes generated into the *business_model*Services project.

The generated classes will always have one attribute for each attribute specified in the class definition, plus an attribute for each association that points to another class.

Custom Generation Options

You access the Custom Generation Options dialog with the File > Custom Generation Options command.

Figure 1-16   Custom Generation Options Dialog
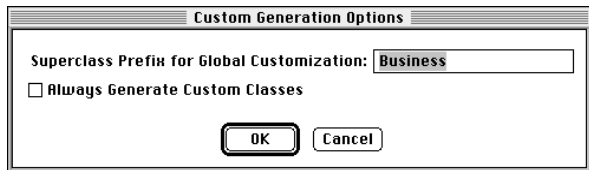
The Superclass Prefix for Global Customization field in the Custom Generation Options dialog specifies the prefix used by the superclasses for the classes generated from the business model. By default, this prefix is "Business," which is the prefix used by the ExpressServices project. You change this value if you customize classes from the ExpressServices project in a new project. Note that you must include the project as a supplier to the business model. See "Global Customization" on page 153 for more information.

Always Generate Custom Classes toggle   When you turn on the Always Generate Custom Classes toggle, Express automatically generates a Base class and a customizable class for every business class. This feature is useful in situations where you know you will be customizing many business classes, because it will be faster to generate all the customizable classes up front, and then simply delete all the customizations on the classes that you do not intend to customize. Note that to delete the class customizations, you must use the Customization Manager, described in "Deleting Customizations" on page 86.

Alternatively, if you do not turn on this toggle, you create a customizable class by selecting the Component > Customize… command for each business class you wish to customize. Express must regenerate the business model the first time you add a customization on a class. Thus, if you plan to customize most of your business classes, it is recommended that you turn on the Always Generate Custom Classes toggle.

Business Class Properties

You access the Business Class Properties dialog by double-clicking a class in the model.

Figure 1-17   Class Properties Dialog
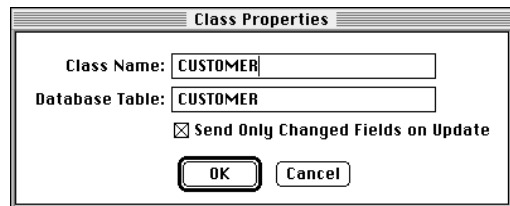
The following properties are set in the Business Class Properties dialog.

Class Name   The class name becomes the prefix for the name of the generated class (for example, CustomerOrderQuery).

Database Table   The database table name is generated into the GetTableName method of the *business_class*BaseQuery class (CustomerOrderBaseQuery) to specify which database table the business class is based on.

Send Only Changed Fields on Update   This property determines the value of the UpdatedClass attribute on the *business_class*BaseQuery (for example, CustomerOrderBaseQuery) class. When the Update method on the BusinessClient class is invoked and the Send Only Changed Fields on Update property is turned off, the UpdatedClass attribute will be set to the BusinessClass object being updated. Otherwise, the attribute is NIL. The Update method determines whether or not to set the UpdatedClass attribute by invoking the SendClassOnUpdate method on the BusinessQuery class. The SendClassOnUpdate method by default returns FALSE. When the Send Only Changed Fields on Update property is turned off, Express generates a SendClassOnUpdate method that returns TRUE.

In the Tutorial example, the Send Only Changed Fields on Update property is on. This means that in an CustomerOrderMgr customization, for example, you will be able to inspect the query to be executed (the CustomerOrderQuery object), but you will not be able to see the CustomerOrderClass object and its values that were used to construct the query.

Association Properties

You access the Association Properties dialog by double-clicking on an association in the model.

Figure 1-18   Association Properties Dialog
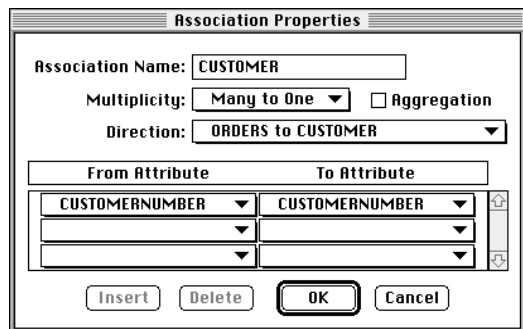
Associations affect what other BusinessMgr objects a given BusinessMgr can reference (by way of its ForeignMgrs attribute).

Association Name   The name of the association becomes the name of the attribute on the generated TOOL class. For example, in the Tutorial business model there is an association called Customer drawn from the CustomerOrder business class to the Customer business class. The generated CustomerOrder class has an attribute called Customer of type CustomerClass. See *A Guide to Express* for information about how Express uses the Direction property to determine on which class to generate the attribute.

Direction   The direction of an association determines which class is the From (referencing) class, and which class is the To (referenced) class. The From class will have an attribute that references the To class. In the example above, CustomerOrder is the From class; Customer is the To class. As a result, the Customer attribute is generated on the CustomerOrder class.

Aggregation   If the Aggregation property is set, then the BusinessMgr of the aggregate class invokes methods on the BusinessMgr of the component class. For example, because there is an aggregate relationship between CustomerOrder and LineItem, the LineItemMgr will be invoked by the CustomerOrderMgr to update data in the LineItem table, while other objects will not interact directly with the LineItemMgr.

Multiplicity   The Multiplicity property controls whether the reference from one class to another is simple (one-to-one, many-to-one), or an array of references (one-to-many). A many-to-one setting indicates that there can be many references to the same object. For example, the association from CustomerOrder to Customer is many-to-one; therefore, the Customer attribute on CustomerOrder is a simple object. Conversely, the association between CustomerOrder and LineItem is one to many; therefore, the LineItem attribute on CustomerOrder is an array of references.

From/To Attributes   Specifies the columns used to construct joins between tables.

Attribute Properties

You access the Attribute Properties dialog by double-clicking a class's attribute in the model.

Figure 1-19   Attribute Properties Dialog
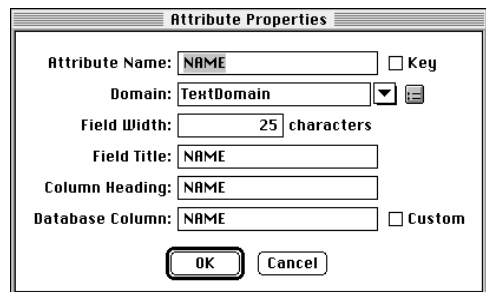
The following properties are set in the Attribute Properties dialog.

Attribute Name   The value for Attribute Name becomes the name of an attribute on the generated BusinessClass object. For example, CustomerClass has the following attributes: Name, Address, Phone, and CustomerNumber.

Key   If an attribute is marked as a key, then Express generates statements for that attribute into the SetKey method on the *business_class*BaseClass. These statements populate the Values attribute on the BusinessKey object.

Domain   The domain will be the data type for the attribute in the generated *business_class*BaseClass.

Field Width   The Field Width value is the width of the field on the generated window.

Field Title   The Field Title text is the field label on a generated form window.

Column Heading   The Column Heading text is the column label on a generated array or outline window.

Database Column   The database column name is generated into the GetColumnName method on the *business_class*BaseQuery class to specify which database column the attribute is associated with.

Custom   This property determines whether the attribute is mapped to a database column. If the Custom toggle is selected, then the attribute will not be part of any queries.

Service Properties

The Service Properties dialog has three tab pages: General, Database, and Connection. You access the Service Properties dialog with the Component > New > Service command.

By default, the general properties tab page appears, shown in Figure 1-20.

Figure 1-20   Service Properties Dialog—General Page
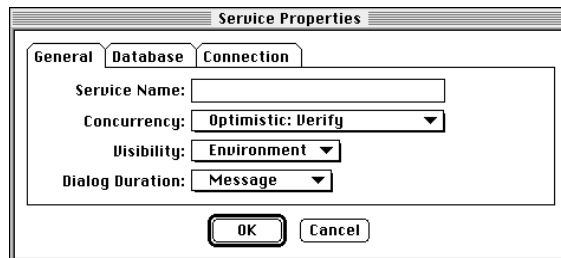
Service Name   The generated DBSession service object and TOOL service object names are based on the Service Name property. For example, for the Tutorial service, Express generates TutorialDBSession and TutorialService.

Concurrency   The value for the Concurrency property is generated into the Setup method of the *business_class*BaseServiceMgr (TutorialBaseServiceMgr), where it is used to initialize values in the ServiceConcurrency object referenced by the ServiceMgr.

The value of the concurrency property also controls whether before images of the BusinessClass objects being updated are sent to the server when the Update method on the BusinessClient class is invoked. If the Concurrency property is Optimistic: Verify, then a copy of the business class before any changes were made (the before image) is stored in the OriginalClass attribute of the BusinessQuery class. Otherwise, the OriginalClass attribute is NIL.

When the Concurrency property is Optimistic: Verify, the Update method of the BusinessDBMgr class uses the OriginalClass attribute to add constraints to the query being run. A constraint is added for each database attribute that is not a LongTextDomain, or any subclass of LongTextDomain. The constraints will cause the query to succeed only if the values of all columns in the database when the update is performed match the values they had when the row that was used to create the BusinessClass object was selected. Database columns used for TextData attributes, or any subclass of TextData such as TextDomain, will have trailing blanks removed before a comparison is made with the BusinessClass attribute (note that the BusinessClass attribute had any trailing blanks removed when it was set). Database columns used for DoubleData attributes, or any subclass of DoubleData, such as DoubleDomain, will be compared for equality within 12 digits of precision.

Visibility   The value for the Visibility property is generated into the TOOL service object based on the *business_model*ServiceMgr class (TutorialService).

Dialog Duration   The value for the Dialog Duration property is generated into the TOOL service object based on the *business_model*ServiceMgr class (TutorialService).

The database properties tab page is shown in Figure 1-21.

Figure 1-21   Service Properties Dialog—Database Page
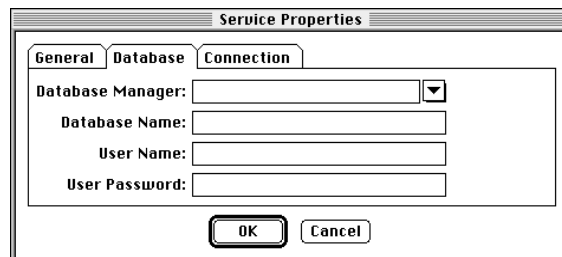
Database Manager   The value for the Database Manager property is generated into the DBSession service object (TutorialDBService) that is generated for each Express service.

Database Name   The value for the Database Name property is generated into the DBSession service object (TutorialDBService) that is generated for each Express service.

User Name   The value for the User Name property is generated into the DBSession service object (TutorialDBService) that is generated for each Express service.

User Password   The value for the Password property is generated into the DBSession service object (TutorialDBService) that is generated for each Express service.

The connection properties tab page, shown in Figure 1-22, allows you to specify options that are processed when the connection to the database is established.

Figure 1-22   Service Properties Dialog—Connection Page
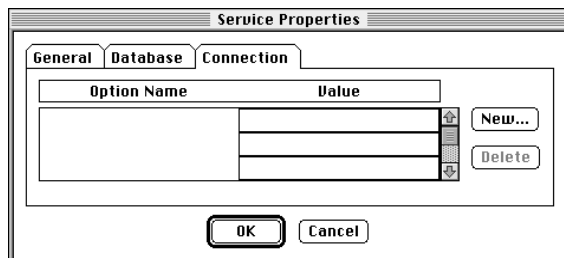
Option Name   The name of the option to be processed when the connection to the database is established. Clicking the New button activates a selection list of options. For a list of available options, refer to the ConnectDB method on the DBResourceMgr class in the *Accessing Databases* manual.

Value   A value appropriate to the specified option.

Application Model Workshop

The TOOL classes derived from a business model will always have one attribute for each attribute specified in the business class, plus an attribute for each association that points to another class. All these attributes will be present in instances of the business class.

However, the attributes that are set by the query are determined by which attributes are displayed as fields in a generated window. In the Application Model Workshop, you have the following options regarding the display of attributes:

• display the attribute

• display the attribute as read only

• do not display the attribute

The first two options will always result in a field on the generated window. If you need to use an attribute in a query, but do not want it displayed on the window, see the Customization Manager Help example, "Selecting a Table Column Not Displayed on Window," which is listed in "Application Model Customization Examples" on page 94. To see this example in context, see the Express example CustomQueryApp; in the CustomQueryAppWindows project, look for the GetRecordTemplate method in the CustomerWindow class.

Application Model Properties

The application model properties are specified in the Application Model Properties dialog, shown in Figure 1-23, below.

Figure 1-23   Application Model Properties Dialog
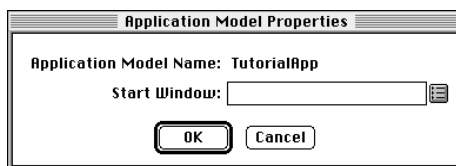
The window specified in the Start Window property is generated into the StartMethod method of the generated project.

Generated Preferences

You set default behavior for generated windows with the Generated Preferences dialog (shown in Figure 1-24). which you access with the File > Generated Preferences command.

Figure 1-24   Generated Preferences Dialog
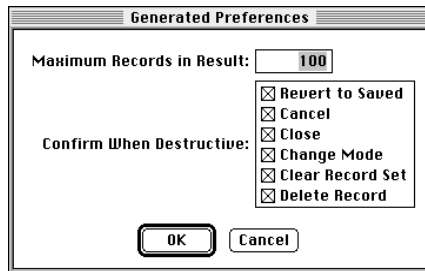
The settings you specify in the Generated Preference dialog are the values of the Preferences attributes in the ApplicationBroker object. Express generates the initial values for the attribute as assignment statements in the *appl_model*BaseBroker's Init method (for example, TutorialAppBaseBroker.Init).

Custom Generation Options

You set custom options for generating client code with the Custom Generation Options dialog, shown in .

Figure 1-25   Custom Generation Options Dialog
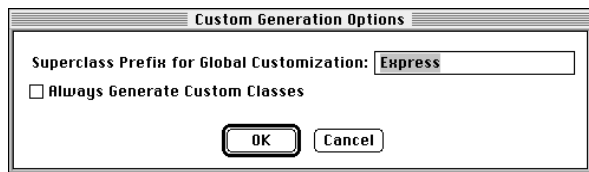
The Superclass Prefix for Global Customization field in the Custom Generation Options dialog specifies the prefix used by the superclasses for the classes generated from the application model. By default, this prefix is "Express," which is the prefix used by the ExpressWindows project. You must change this value if you customize classes from the ExpressWindows project in another. Note that you must include the project as a supplier to the application model. See .

Always Generate Custom Classes toggle   When you turn on the Always Generate Custom Classes toggle, Express automatically generates a Base class and a customizable class for every window class. This feature is useful in situations where you know you will be customizing many window classes, because it will be

faster to generate all the customizable classes up front, and then simply delete all the customizations on the classes that you do not intend to customize. Note that to delete the class customizations, you must use the Customization Manager, described in "Deleting Customizations" on page 86.

Alternatively, if you do not turn on this toggle, you create a customizable class by selecting the Component > Customize… command for each window class you wish to customize. Express must regenerate the application model the first time you add a customization on a class. Thus, if you plan to customize most of your window classes, it is recommended that you turn on the Always Generate Custom Classes toggle.

BusinessClass Window Properties

Double-click on a BusinessClass window in the drawing area of the Application Model Workshop to access its property sheet, shown in Figure 1-26, below.

Figure 1-26   Business Class Window Properties Dialog
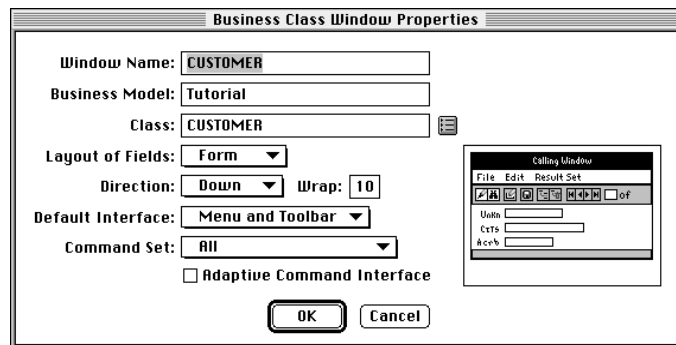
Window Name   Express appends "Window" to the Window Name value. This name is the name of the generated window class and appears in the title bar of the generated window.

Layout of Fields   Express uses the value of the Layout of Fields property to determine which class will be the superclass of the generated *business_class*Window (or, if customized, *business_class*BaseWindow) class (for example, CUSTOMERORDERBaseWindow). The superclass is determined as follows:

Layout of Fields Value	Superclass
Form	ExpressFormWindow

Layout of Fields Value	Superclass
Array	ExpressArrayWindow
Outline	ExpressOutlineWindow

Direction The value for the Direction property affects the generated classes only if the Layout of Fields property is Form. A setting of Down generates fields below the previous field and a setting of Across generates fields to the right of the previous field.

Wrap The value for the Wrap property affects the generated classes only if the Layout of Fields property is Form. The specified number tells Express how many fields to generate before starting a new row or column of fields.

Default Interface The Default Interface property affects the type of interface commands generated onto the window and displayed at runtime. If the Default interface is set to Menu and Toolbar, then Express generates two attributes in the CommandMgr object: MenuSet, which points to a MenuSetDesc object, and ToolbarSet, which points to a ToolBarSetDesc object, and displays a menu and toolbar on the window at runtime.

Note that this property is overridden at runtime if the Adaptive Command Interface property is set.

Command Set The Command Set property determines which of the commands are generated onto the window. The Default Interface property determines how those commands appear to the user.

Command Set	Generated Commands
All	Commands that allow both search and update of records (includes scrolling commands).
Search	Commands that allow a user to retrieve records (includes scrolling commands).
Multiple Record Edit	Commands that allow update of multiple records (includes scrolling commands).
Single Record Edit	Commands that allow update of a single record.
Multiple Record View	Commands that allow users to view of multiple records (includes scrolling commands).
Single Record View	Commands that allow users to view a single record.

Note that this property is ignored if the Adaptive Command Interface property is set.

Adaptive Command Interface   The Adaptive Command Interface property determines whether the Default Interface and Command Set properties are used to restrict which commands are generated onto the window, when the window is called from another Express window, and whether the calling window has control over what commands the window displays. If the Adaptive Command Interface property is set, then the CommandSet and CommandInterface attributes in the LinkInfo object are set by a combination of factors in the business model and Link Properties dialog that indicate the appropriate command set and interface. Otherwise, those attributes are determined by the Default Interface and Command Set properties in the Link Properties dialog.

Link Properties

Double-click on a link in the drawing area of the Application Model Workshop to access its property sheet, shown in Figure 1-27, below.

Figure 1-27   Link Properties Dialog
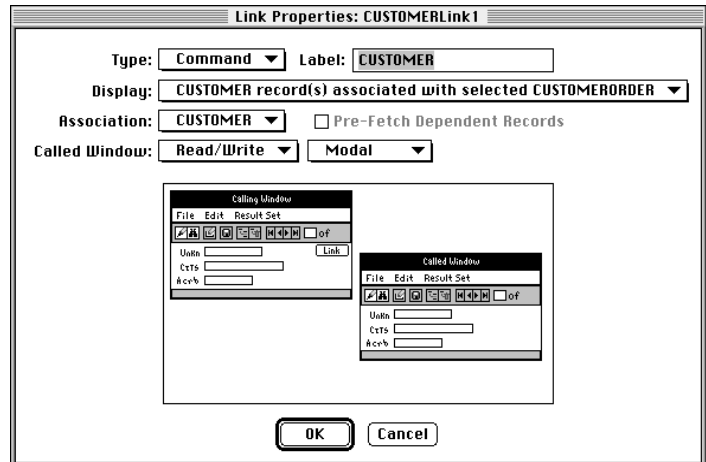
Type   The Type property determines what widgets Express generates onto a window, and also what link methods are generated into the *business_class*BaseWindow class. The table lists the widgets associated with each link and the generated link methods. See the iPlanet UDS online Help for descriptions of the generated methods.

Link	Widgets	Generated Methods
Command	Push button	CommandLinKTo*link_name*Link#
Drilldown	none	DrillDownLink
Folder	FolderTab	*InitFolderlink_name*Link#
Lookup	Push button	LookupLinkTo*link_name*Link#
Nested	Placeholder panel	InitNestedLink*link_name*Link#

Display   Controls which records to display in the generated window. Also affects the code in the generated methods, such as CommandLinkTo, and so on.

Association   This property is used when there are multiple associations between business classes to determine which association to use. The association affects the SQL statements generated at runtime.

Pre-Fetch Dependent Records   The Pre-Fetch Dependent Records property determines how Express selects detail data. Note that the Pre-Fetch Dependent Records property is available only for a window class that is based on a business class that is a "from-class" in a business model. The following table describes how Express retrieves detail data depending on the status of the Pre-Fetch Dependent Records property:

Pre-Fetch Dependent Record Property	Express Behavior
Not settable	There is no attribute in the calling window in which to save the detail data, so Express reselects detail records each time the user scrolls to another master record. Express reselects the data even if the user scrolls to a record whose detail data has already been retrieved.

Pre-Fetch Dependent Record Property	Express Behavior
Available, but not on	Each time the user scrolls to a new master record, Express passes a *business_class*BaseQuery object to execute the query based on the criteria stored in the InitialSearch attribute in the LinkInfo object. The records are retained on the client, and Express does not re-execute the query each time the user scrolls to the same data for viewing.
On	Express selects all the detail records for each master record, and passes the entire result set to the called window in the ResultSet attribute in the LinkInfo object.

Called Window

Mode status   The mode status setting determines if the end user must exit the called window before he or she can access other windows in their application. The Mode status property determines the value of the LinkStyle attribute on the LinkInfo class.

Mode Status Setting	LinkStyle Attribute Value
Modal	WS_MODAL
Modeless	WS_MODELESS

Read status   The Read status property determines the value of the IsReadOnlyResultSet attribute in the LinkInfo object.

Read Status Setting	IsReadOnlyResultSet Attribute Value
Read/Write	FALSE
Read Only	TRUE

Callout Properties

You access callout properties with the Component > New > Callout command, which displays the New Callout dialog:
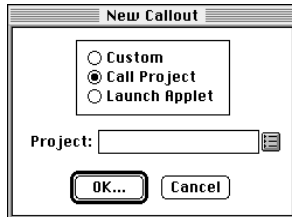
Selecting any type of callout and clicking OK brings up the Callout Properties dialog, shown in Figure 1-28.

Figure 1-28   Callout Properties Dialog
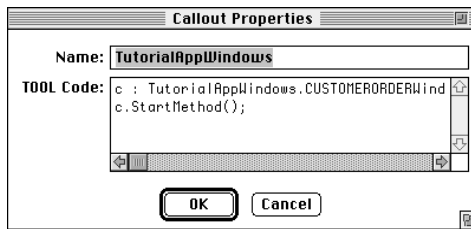
Name   For a Custom or "Call Project" callout, the Name property will become the name of the method generated for the link to the callout. For a "Launch Applet" callout, it will be the name of the applet.

TOOL Code   The TOOL code you enter will become the code in the command link method in the generated read-only window class of the window that invokes the callout (the read-only class is *business_class*Window if there are no customizations, *business_class*BaseWindow if there are).

Customizing Express Applications

In general, you customize an application by using the Customization Manager. The Customization Manager provides a set of common customizations

Topics covered in this chapter include:

- using the Customization Manager

- creating customizable classes

- deleting customizations and customizable classes

- online customization examples

- customizing generated window classes

- adding business rules to a client

- manipulating result sets

- customizing queries

- global customizations

Note that the examples and illustrations in this chapter use the example created in *A Guide to Express*, and make references to the class and model names used there. Also, this chapter assumes some familiarity with Chapter 1, "Express Application Architecture."

Overview

Read this overview to learn some basic facts about customizing Express applications. Before making Express customizations, make sure that the behavior you want cannot be accomplished by simply setting options within the Application or Business Model Workshop and regenerating.

General Considerations

The following are suggestions you should take into consideration before you begin your customizations.

- Use the Customization Manager (see "Customizing With the Customization Manager" on page 81) if possible, rather than create customizations directly in generated projects.

- You can choose to customize specific classes, or make customizations to your application as a whole by modifying classes that are not represented in the model (for example, *applicationName*Broker or ServiceMgr).

- Always try to make your customizations in such a way that they continue to work after you later make changes to and regenerate your application model. For example, if you change your Layout of Fields property from Outline to Array, you want your customizations to continue to work.

- Before customizing, first try to add and remove components from generated windows by changing window and link properties in the Application Model Workshop. If you cannot achieve what you need to in the workshop, see "Setting Widget State" on page 104.

- When adding a field to a generated window, remember to use "new()" to create it in the Init method and to call super.Init in the Init method.

- It is best not to create new classes in projects generated by Express. Try to keep to only generated classes in these projects (you can of course customize these classes—just do not add new ones).

- Decide whether you need to customize a few classes or many. Your decision will determine whether or not to turn on the Always Generate Custom Classes toggle. When turned on, Express automatically generates a "Base" class and a leaf-level (customizable) class for each business or window class in the model. The downside is that you have to delete any unwanted customizable classes one by one, using the Customization Manager.

Creating Customizable Classes

Express generates class components into the leaf-level classes, which are not customizable. This creates the minimum number of classes required by the application, and therefore the smallest image size for deployed applications. To customize an application requires that you first create customizable classes. You do

this either by creating individual customizable classes with the Customization Manager, or by creating customizable classes for all components in the model using the Always Generate Custom Classes option on the Custom Generation Options dialog.

When you create customizable classes—whether a single class or one for every class in the model—Express automatically expands the class hierarchy, creating a Base class above the leaf-level class. All the components that were generated into the leaf-level class are moved to the Base class, leaving the leaf-level class free for customizations. The Base class is renamed to *business_class*BaseClass (or *appl_model*BaseWindow), and the new, customizable class is named *business_class*Class (or *appl_model*Window). Figure 2-1 illustrates this principle.

Figure 2-1   Naming Conventions Before and After Creating Customizable Classes
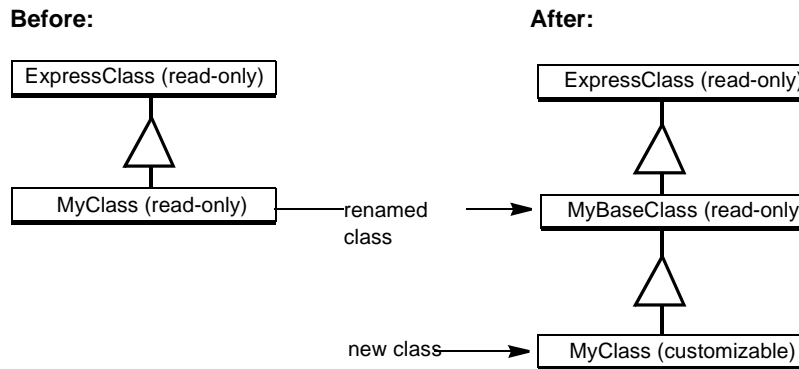
If you decide you do not want the customizable class, you delete it using the Customization Manager (see "Deleting Customizations" on page 86). This collapses the three-class hierarchy back into the two-class hierarchy and renames the read-only Base class to its original name.

Creating a Single Customizable Class

The Customization Manager allows you to customize individual classes. When you invoke the Customization Manager for the first time on a specific class, Express automatically expands the class hierarchy for the particular class, as shown in Figure 2-1. For more information, see "Customizing With the Customization Manager" on page 81.

Creating Customizable Classes for All Classes

If you know you will be customizing many classes, it might be more convenient to generate the hierarchy at the onset of your development cycle. You do this by setting the Always Generate Custom Classes toggle in the Custom Generation Options dialog. When set, this option tells Express to automatically create the full hierarchy for every business or window class in your model. When you use this option, note:

- Turning it on turns it on for all the classes in the model (business classes in the business model or window classes in the application model, depending on where you set the option); it can then only be turned off by deleting all the customizations on each class using the Customization Manager. You should not delete the individual classes in the generated project.

    In other words, you create customizable classes for all business or window classes in the model with one step, but you must remove individual customizable classes separately.

- Turning it off will only affect new classes created from the time you turned it off, resulting in some Base classes having "Base" in their names and some not.

➤ **To create customizable subclasses for every business or window class in your model**

1. In the Business Model or Application Model workshop (depending on where the classes are that you want to customize), choose the File > Custom Generation Options… command.

    The Custom Generation Options dialog appears.:

2. Turn on the Always Generate Custom Classes toggle.

    This toggle causes Express to generate customizable leaf-level classes for every class in the business or application model.

3. Click OK.

4. Generate code.

In the Business Model, choose the File > Generate Server Code command.

In the Application Model, choose the File > Generate Client Code command.

See "Deleting Customizations" on page 86, for information on deleting customizable classes.

Customizing With the Customization Manager

The Customization Manager assists you in customizing your Express applications. When you start the Customization Manager on a class, you are presented with a list of common customizations. When you double-click on one of the customizations, the Customization Manager automatically opens a Method Workshop window, displaying the appropriate method where you will place your customization code, as shown in Figure 2-2. The initial method code contains documentation of the method parameters and return code, and, if required, a super.*method* statement and a return statement.

Figure 2-2   Customization Manager Opening the Appropriate Method Workshop
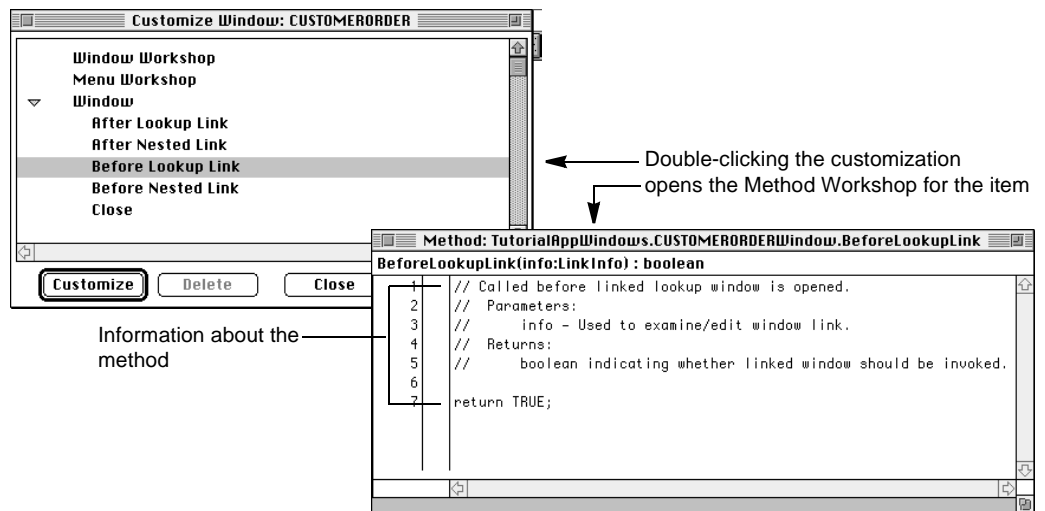
In addition to locating your customization code for you, the Customization Manager has online help associated with each customization, including example code, and in many cases several related examples. which you can copy and paste into the Method Workshop to give you a start on your customization code. You can access on help by pressing the Help button while a customization is selected, as shown in Figure 2-3.

Figure 2-3   Customization Manager's Help on a Customization Topic
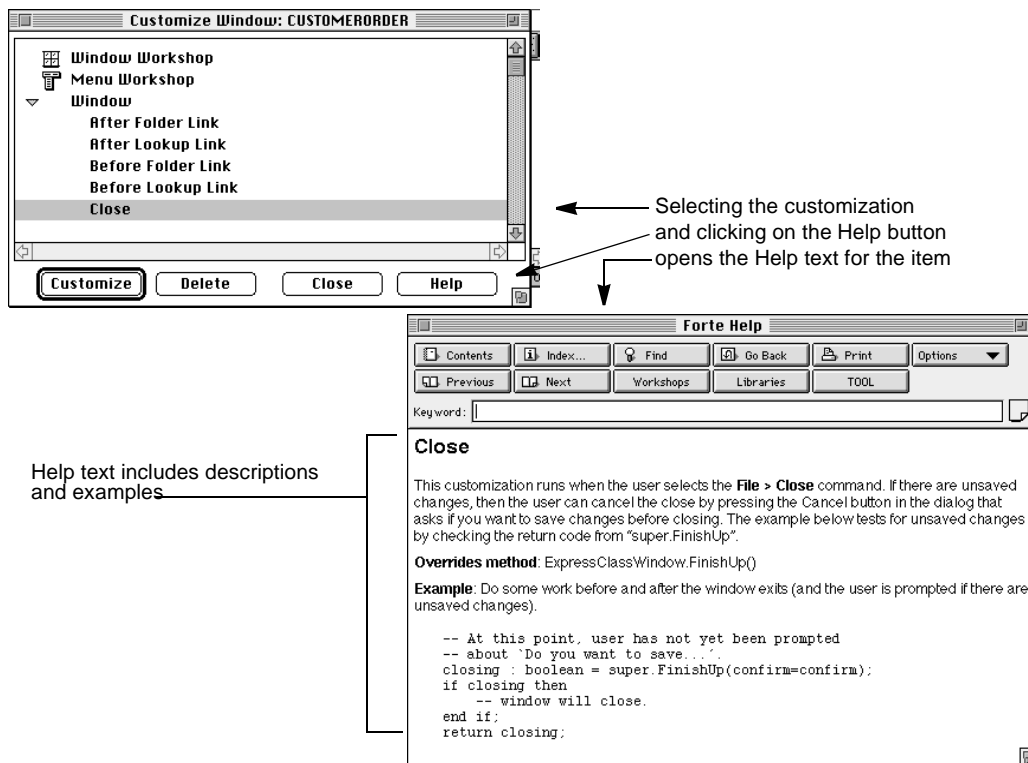
A comprehensive list of examples available from the Customization Manager is provided in "A Roadmap to Customization Examples" on page 91.

You also use the Customization Manager to delete customizations. You can delete specific customizations, or entire subclasses. (You delete a subclass by deleting all its customizations, an operation that deletes the subclass, renames the Base class, and collapses the hierarchy to the "Before" structure in Figure 2-1 on page 79.) You can also delete window customizations, reverting back to the original window while preserving the rest of the window class's customizations.

The following sections describe how to use the Customization Manager.

Using the Customization Manager

You use the Customization Manager to customize specific classes in your model. Any instructions for using the Customization Manager apply to both the Business Model and Application Model Workshops.

When you start the Customization Manager for a read-only class, a customizable subclass is created for the class. Figure 2-1 on page 79 shows the naming conventions for the new class hierarchy.

➤ **To customize a class using the Customization Manager**

1. Select the window class or business class you wish to customize.

2. Choose the Component > Customize… command.

   The Customization Manager appears.
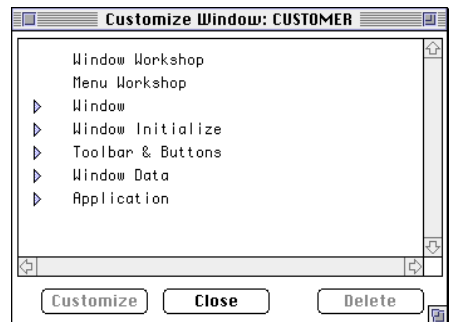
3. Click the arrow to the left of the categories to view specific customizations.
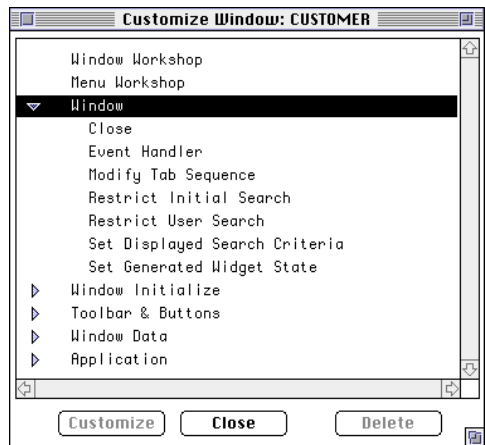
4. (Optional) Select the customization you wish to perform and press the Help key.

A help screen appears as shown in . If you like, you can copy the code into the clipboard before you open the Method Workshop. For a complete description of the examples available through Customization Manager Help, see .

5. Double-click the specific customization you wish to make.

If this is the first customization you are making to this class, Express displays the following dialog:
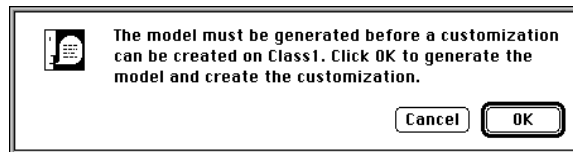
6. Click OK.

    Express automatically expands the class hierarchy to include a customizable class for the selected window or business class, and the Method Workshop opens, displaying an override of the appropriate method.

    ```
    ┌─────────────────────────────────────────────────────────────┐
    │ ▣▭▭▭▭ Method: MyAppModelWindows.CUSTOMERWindow.FinishUp ▭▭▭ ▣▮│
    │ FinishUp(confirm:boolean=FALSE) : boolean                    │
    │   1 │   return super.FinishUp(confirm=confirm);           ⇧  │
    │     │                                                        │
    │     │                                                        │
    │     │                                                        │
    │     │                                                        │
    │     │                                                        │
    │     │                                                        │
    │     │                                                        │
    │     │                                                     ⇩  │
    │     │◁                                              ⇨        │
    └─────────────────────────────────────────────────────────────┘
    ```
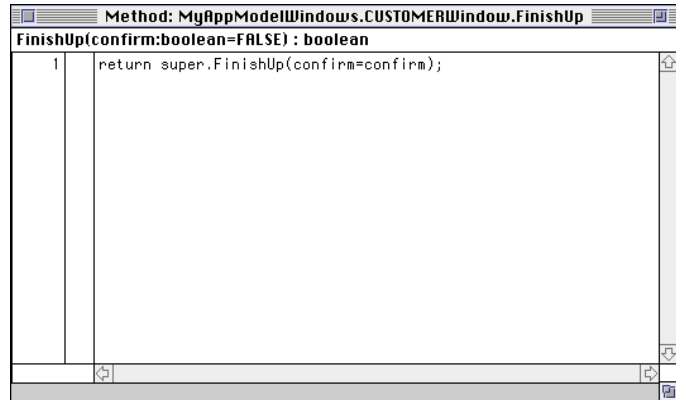
7. If you have copied code from the online help, you can paste it into the Method Workshop.

    Make sure you remove the default call to super.*methodname*, as the help example code already includes it. Make any changes or additions required for your specific customization needs.

8. Choose the File > Compile command to compile the method, which determines whether there are errors.

9. (Optional) Return to Step 3 and add more customizations to the class.

10. Close the Method Workshop and the Customization Manager window.

Note that the Customization Manager window is not modal. In other words, you can view customization information about a class, leave the window open and select another window or business class, and the Customization Manager will display the appropriate information for the newly selected class.

Deleting Customizations

You can use the Customization Manager to delete specific customizations or to delete all customizations in a class. If you delete all customizations, the Customization Manager deletes the customizable class, renames the Base class to its original name, and collapses the class hierarchy (see the "Before" structure in Figure 2-1 on page 79).

NOTE	If you toggled the Always Generate Custom Classes toggle on in the Custom Generation Options dialog to generate a complete hierarchy of customizable classes, you must toggle it off before deleting any classes.

Deleting Specific Customizations

You can tell that a method has been customized by the method symbol next to the customization, as shown below:

Figure 2-4   A Select Customization Exists
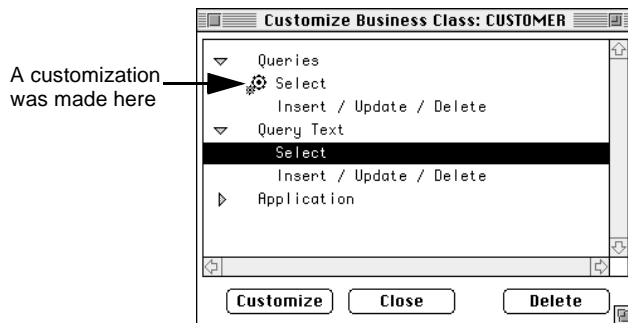
NOTE	Several of the customizations map to the same method. If you customize a method that is mapped to multiple customizations, the symbol will appear beside both customizations. For example, two customizations in the Application Model (Window/Set Generated Window State and Window/Set Displayed Search Criteria) both map to the ClearFieldsForSearch method. If you create one of these customizations, the symbol appears on the other, as well. In fact, if you create a ClearFieldsForSearch method outside the Customization Manager, the symbol will appear beside both these customizations. When you delete the customization you implemented, the symbol disappears in all locations.

➤ **To delete a customization**

1. Select the class that contains the customization and choose Component > Customize….

   The Customization Manager appears.
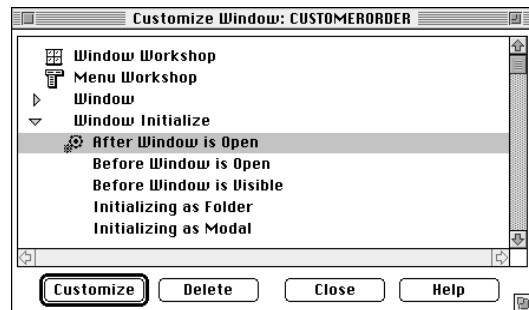
2. Select the customization you wish to delete and click the Delete button.

   The Delete Customization dialog appears.

3. Turn on the Delete Selected Customization toggle and click OK.

   Express deletes the selected customization.

Deleting a Class

When you delete all customizations for a class, Express removes any customizations in the leaf-level class, moves the class components from the leaf class to the Base class, removes the leaf class, and renames the Base class to its original name.

Note that each time you Delete All Customizations, Express regenerates the model. If you plan to delete several classes in a model by deleting all customizations to those classes, you can defer the model regeneration until you specifically request it.

➤ **To delete all customizations (entire class)**

1. If the Always Generate Custom Classes toggle is turned on, turn it off.

   If this option is left on, you will recreate any classes you delete the next time you generate code. Access this toggle by choosing the File > Custom Generation Options… command.

2. Select the class you wish to delete and choose Component > Customize…

   The Customization Manager appears.
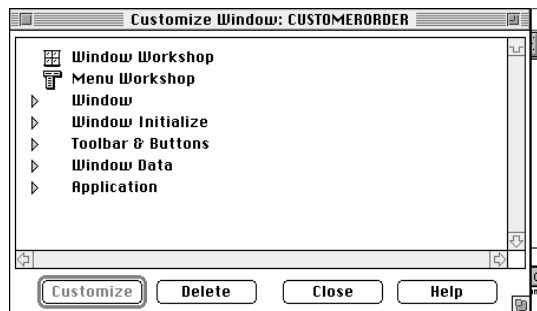
3. Click the Delete button.

   The Delete Customization dialog appears, with only the Delete All Customizations toggle active.
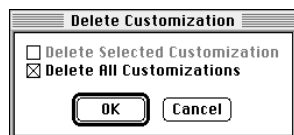
4. Click OK.

   The following dialog appears.

When To Code Generate
Deleting all customizations on CUSTOMERORDER requires a Code Generation. Click "Generate Now" to Code Generate now and delete all customizations; click "Defer" to delete the customizations next time you do a Code Generation.
Generate Now   Defer   Cancel

5. Click Defer or Generate Now, depending on whether you have more classes to delete (Defer) or not (Generate Now).

   See the next section for information on deferring generation.

   When you choose Generate Now, Express removes any customizations (if any) in the leaf-level class, and returns the class hierarchy to its original "Before" structure shown in Figure 2-1 on page 79.

Deferred Deletion of Customizations

Each time you delete All Customizations for a class, Express automatically regenerates the model. If you plan to delete all customizations in multiple classes, you can use the Defer button in the Delete Customization dialog to defer regeneration of the model until you explicitly choose to regenerate. For example, say you wish to delete all customizations for three business classes. For the first two classes, choose the Defer button in the confirmation dialog. When you delete all customizations for the third class, choose the OK button. This will automatically cause Express to regenerate the model. Alternatively, you can also choose the File > Generate (Client/Server) Code command. Note that if you exit the workshop without regenerating the model, the customizations will *not* be deleted, and you will be prompted as a reminder that the Delete All Customizations that you specified will not occur.

➤ **To delete all customizations and defer the deletion**

1. Perform steps 1 - 4 above.

2. Click the Defer button.

NOTE	You must regenerate the model before leaving the Business/Application Model Workshop or the class(es) will not be deleted.

Deleting Window and Menu Customizations

You make window and menu customizations in the Window and Menu Workshops, respectively. Express can automatically delete window and menu customizations, while preserving other types of customizations.

➤ **To delete a window or menu customization**

1. Select the window class whose window or menu customizations you wish to delete and choose the Component > Customize… command.

    The Customization Manager appears.
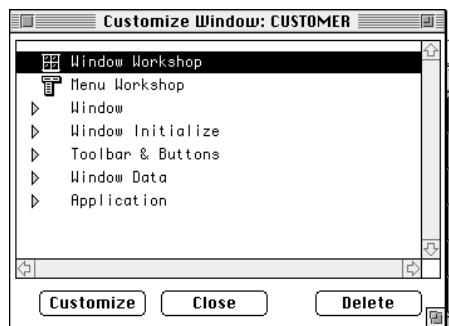
2. Select Window Workshop (or Menu Workshop) and click the Delete button.

    The Delete Customization dialog appears.

3. Turn on the Delete Selected Customization toggle and click OK.

    Express deletes the all window (or menu) customizations.

NOTE	You can only delete window or menu customizations using the Customization Manager.

Application-Wide Customizations

The Customization Managers for both the Business Model and Application Model Workshops have a customization category of "Application." Application customizations are customizations that affect the application as a whole, rather than a specific window or business class in the model, such as Exit and Start.

Deleting application customizations   When you delete all customizations for a business or window class, application customizations are not deleted. You must select each application customization specifically and then delete it.

A Roadmap to Customization Examples

Express provides many customization examples and tools to help you customize your application. These are provided either in printed form (in this chapter) or are accessible through the Customization Manager online Help. The longer examples are both online and printed in this chapter.

Customization examples fall into these categories:

- Customization Manager Help files—online only

  A complete list is provided in "Business Model Customization Examples" on page 93 and "Application Model Customization Examples" on page 94.

- complex examples—both online and in this chapter

  A complete list is provided in "Complex Examples" on page 97.

- customization techniques—in this chapter

  Use the Table of Contents in this manual to help locate the sections in this chapter that describe the type of customization you wish to make.

- example Express Applications—shipped with Express

  For a complete description of these example applications, see Appendix A, "Express Example Applications."

Customization Manager Help Files

Using the Customization Manager is fully described in the previous section, "Customizing With the Customization Manager" on page 81.

At any point while you are using the Customization Manager, you can press the Help key to display online information about the currently selected customization.

Figure 2-5   Customization Manager Help—Application Model Workshop
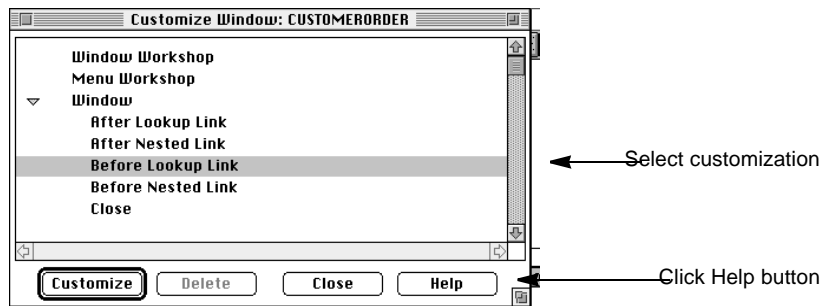
There is example code for most customizations, which you can copy and paste into the Method Workshop to give you a start on your customization code. If you paste example code into the Method Workshop, be sure to delete the default override of super.*method*, as that method invocation also appears in the example code.

You can also access these item-specific customization examples directly by their titles on the List of Examples in the top Customization Manager window (when you first open it and nothing is selected), or by the List of Examples link on every Help topic page. Complex examples (see "Complex Examples" on page 97) are included in the list.

Business Model Customization Examples

The table below lists all Help examples available from the Customization Manager when you access it from the Business Model Workshop.

Customization Section	Customization	Example Title
Queries	Generate New Key Value	Creating a New Key for a BusinessClass Object
	Select	Adding a Restriction to a Select Query
		Examining a Select Query
	Insert/Update/Delete	Validating an Attribute
		Validating a Record
		Substituting an Update for a Delete
	Insert/Update/Delete	Adding a Customer Number to a Record Without Displaying It
Query Text	Select	Dump Select Queries if Log Flags are Set
		Check If Query Contains a Where Clause
	Insert/Update/Delete	Writing Generated SQL to a Log File
Application	Before Select	Checking for Queries on a Particular Business Class
	After Select	Setting the Value of a Derived Field
	Before Update	Checking for Inserts on a Particular Table
	After Update	Auditing Modified Values

Application Model Customization Examples

The table below lists all Help examples available from the Customization Manager when you access it from the Application Model Workshop.

Table 2-1   Help Examples Available from the Customization Manager

Customization Section	Customization	Example Title
Window	After Command Link	After Exiting a Window Called by a Command Link
	After DrillDown Link	After Exiting a Window Called by a DrillDown Link
	After Folder Link	After a Folder Window Is Initialized
	After Lookup Link	After Exiting a Window Called by a Lookup Link
	After Modeless Command Link to <name> Open	After a Modeless Window to a Command Link Opens
	After Modeless DrillDown Link to <name> Open	Saving a Reference to a Modeless Window
	After Nested Link	After a Nested Window Is Initialized
	Before Command Link	Before Invoking a Window Called by a Command Link
	Before DrillDown Link	Before Invoking a Window Called by a DrillDown Link
	Before Folder Link	Before a Folder Window Is Initialized
	Before Lookup Link	Before Invoking a Window Called by a Lookup Link
	Before Nested Link	Before a Nested Window Is Initialized
	Close	Closing the Current Window
	Event Handler	Handling Mouse Clicking Events
		Updating Records in Event Code
		Handling Row Exit Events
	Folder Tab Selected	When a Folder Tab Is Selected

Table 2-1 Help Examples Available from the Customization Manager *(Continued)*

Customization Section	Customization	Example Title
Window (*continued*)	Modify Tab Sequence	Adding a Field to the End of a Tab Sequence
		Adding a Field on a Nested Window to a Tab Sequence
		Adding a Field to the Middle of a Tab Sequence
	Modify Tab Sequence	Adding a Field to the Start of a Tab Sequence
		Removing a Field from a Tab Sequence
	Restrict Initial Search	Restricting Search Criteria From a Calling Window
	Restrict User Search	Restricting Search Criteria From a User
	Set Displayed Search Criteria	Setting Displayed Search Criteria
	Set Generated Widget State	Setting a Generated Widget State
Window Initialize	After Window is Open	Setting a Window to Search or Edit Mode
	Before Window is Visible	After a Window is Initialized but Before It is Visible
		Changing a Window's Title
	Before Window is Open	Restricting a Window's Initialization Query
		Setting a Window's Initial Query to Select All Records
	Initializing as Folder	Customization Called Only When Window is a Folder
	Initializing as Modal	Positioning a Modal Window
	Initializing as Modeless	Customization Called Only When Window is Modeless
	Initializing as Nested	Checking If a Window Is Nested

Table 2-1   Help Examples Available from the Customization Manager *(Continued)*

Customization Section	Customization	Example Title
Toolbar & Buttons	Check Edit or Search Mode	Checking Whether Edit or Search Mode Was Selected
	Delete Record from Result Set	When Deleting a Row from a Result Set
	Insert Record into Result Set	When Inserting a Row into a Result Set
	Save	Checking for User Changes Before a Save
		Validating Data Before a Save
	Search	Adding Processing Before/After Search
Window Data	Add Records to Save	Adding Related Records to a Result Set Before a Save
	Create New *myClass*Class	Customizing New BusinessClass Objects
	Field Value Changed	Calculating a Derived Field
		Posting an Event on a Changed Field Value
	Record Displayed	Customizing Events Based on Displayed Records
	Display Current Record	Printing a Message on a Record Select
		Executing a Query on a Record Select
	Select Not-Displayed Table Column	Selecting a Table Column Not Displayed on Window
	Set Values in New Record	Setting Default Values in Newly Inserted Records
		Inserting a Record into a Result Set
	Validate Fields	Validating Fields
		Validating Domains
		Validating a Lookup in an Array Field
	Validate Records	Validating Records
		Checking That Data is Entered in Mandatory Fields
	Validate Record -- Mark Valid/Invalid	Validating Whether a Record Needs to Be Validated

Table 2-1   Help Examples Available from the Customization Manager *(Continued)*

Customization Section	Customization	Example Title
Application	Exit	Customizing an Application at Exit
	Start	Using a Non-Express Window As a Login Window

Complex Examples

Several complex customization examples are available, both through the Customization Manager's online Help and printed in this chapter. These are listed in the following table

Location in Customization Mgr	See...	Example Title
List of Examples	page 129	Using an Express Window as a Login Window
Hot link on: Window Data/Field Value Changed	page 133	Calculating a Derived Field From Nested Window Data
Hot link on: Window Initialize/Before Window is Visible Window Data/Add Records to Save	page 136	Generating Records with Unique Sequence IDs
List of Examples	page 138	Synchronizing Data in a Modeless Linked Window
List of Examples	page 140	Customizing the Database Mapping of a Business Class

Syntax of Examples

The examples throughout this document and in the Express Help system are a mixture of complete methods and code fragments from larger methods. The method name and parameters that you are to specify in the Method Properties dialog are identified in this manual as bolded text at the top of the code. In the Help text, a line above the code that begins Overrides Method: identifies the method. Note that parameters are *input* parameters unless otherwise stated. The example below illustrates a method definition in bold, followed by method code:

```
method myWindow.Search(query : BusinessQuery) :
      Array of BusinessClass

-- put Before-Search code here.

data : Array of BusinessClass =
         Super.Search(queryTree);

if (GetCurrentRecord().InstanceStatus < BusinessClass.ST_EMPTY)
    then
      -- put After-Search code here.

end if;

return data;
```

Customizing Manually

You can perform customizations that do not appear in the predefined list of common customizations in the Customization Manager. To do so, open the project that contains the class you wish to customize and then override the appropriate method or methods in that class. However, before you can customize a class, the class hierarchy must include the customizable leaf-level class. These are always classes whose superclass name contains the word "Base." As stated previously, these are not automatically generated by default. To create customizable classes, see either "Creating a Single Customizable Class" on page 79 and "Creating Customizable Classes for All Classes" on page 80.

Customizable classes are never regenerated, so any changes you make to them will be preserved. For window classes, if you modify the window associated with the class, then those changes are merged into the customizable window according to the rules of window inheritance (see *A Guide to the iPlanet UDS Workshops* for information about window inheritance).

The most common way to customize the behavior of an Express application is to override a method in a superclass with TOOL code of your own.

Locating Where to Customize

Making customizations through the Customization Manager automatically locates them where you want them. If none of the customizations provided by the Customization Manager suit your needs, then this section will help you locate common places to add customizations. If the type of customization you want to do is not covered here, follow these suggestions:

• Look through the runtime scenarios in Chapter 1, "Express Application Architecture." If processing similar to what you need is covered, then you will probably see which method you need to override.

• Review the OMT class descriptions in Chapter 1, "Express Application Architecture." The interconnections between objects can be useful in figuring out where to make a customization. The class descriptions following each class diagram can also be useful.

• Run your application under the Debugger to just before where you want to change behavior. Set "Post" and "When" breakpoints in the Debugger and continue execution of your application. Step in from there until you locate the best method to override.

In addition, you can examine inherited class elements to help you determine what and where to customize.

Overriding Methods in a Superclass

Whenever this document instructs you to override a method, you must include a call to the method in the superclass (super.*method*) in your customized code. Omitting this call in most cases will cause the overridden method to fail to perform properly. In rare cases, you must *not* include a call to super.*method*—in these cases, the instructions will point this out explicitly. Customizations provided by the Customization Manager create initial code for the method that has a super.*method* method, if one is required.

You *override* an existing method by creating a new method in the customizable class (for example CUSTOMERORDERWindow) identical in name, parameters, and return values to the method in the superclass (CUSTOMERORDERBaseClass). Then, invoke the superclass method (to access its functionality) and add your own custom TOOL code.

When you want to create a method in a subclass that overrides a method defined in a superclass, drag the method from the Class Workshop for the superclass to the Class Workshop for the subclass.

➤ **To override a method**

1. Select the Inherited command from the View menu in the Class Workshop to help you find which methods are defined in a superclass.

2. Use the Open SuperClass command from the File menu in the Class Workshop to locate the superclass where the method is defined.

3. Drag-and-drop the method from the superclass to your customizable subclass.

   This will create a method in the subclass with the correct name, parameters, and return type.

4. Open the method in the subclass and delete all its code, replacing it with the single statement:

   ```
   return super.method_name(parameter_list);
   ```

   For example, when overriding a method called Search, which has a parameter named queryTree and returns a value, replace the code in the newly created subclass method with the following statement:

   ```
   return super.Search(queryTree=queryTree);
   ```

5. Add your custom code in this overridden method.

If you were to override a superclass method by dragging the method into the subclass and then modifying the code, you would not need to call super.method. However, do not customize in this manner, because future versions of Express may change the implementation of the original method, causing your customized method to fail to compile or to execute improperly. Your call to super.method encapsulates the method's behavior, making Express upgrades simpler.

Customized Event Handling

To handle events in your windows, use the Customization Manager's Window/Event Handler customization, which creates an event handler named CustomEvents. Do not try to add event handling by creating an event loop in a Display method, as you would in a non-Express window. When CustomEvents is present, Express invokes this handler the WindowEvents or NestedWindowEvents event handlers in the *business_class*BaseWindow. Events in the CustomEvents handler will be added to the event loop.

If your CustomEvents event handler registers for an event that the Express-generated classes also register for, then your registration will override the Express registration (your event is registered last, and according to the rules for event registration, the last registration for a particular event takes effect). In general, Express will register for the following events on the generated buttons, menu items and fields:

- a Click event on all buttons

- an Activate event on all menu items

- an AfterValueChange event on all data fields

- a ChildAfterFirstKeyStroke on the <DataGrid> grid field

- an AfterRowValueChange and an AfterRowEntry on all array fields

- an AfterCurrentNodeChange on all outline fields

If you also need to perform some operations within one of these events, then you will need to look at the Express TOOL code for that event in the *business_class*BaseWindow or its superclasses and find a method to override. Several of the examples in this chapter override methods invoked in response to one of these events (for example, ValidateField, FieldValueChanged, or AfterChildWindowChange).

Explicitly Posted Events

You may post and handle your own events in Express-generated windows, without restriction. However, you should not try to handle events that Express posts—Express needs to handle these. See "Calculating a Derived Field From Nested Window Data" on page 133 for an example that posts events between windows.

Local and Global Customizations

Most customizations affect a particular class or window and are thus "local" in nature. However, you may need to make some customizations that will affect all windows or classes (for example, add a button or logo to all windows). The coding techniques for making local and global customizations are similar, but special steps must be taken to cause a change to affect all future generated windows or classes. These steps are described in "Global Customization" on page 153.

Error Reporting

You can handle errors and raise exceptions in your customizations as you would in a non-Express application (using the GetTextData method on the MsgCatalog class, the AddError method on the ErrorMgr class, and so on). See the *TOOL Reference Guide* and Framework Library online Help for more information.

You will see that classes in the ExpressServices and ExpressWindows projects make use of an Error class defined in ExpressServices, but this class is not intended for you to customize or subclass. Most uses of the Error class are in raise statements, like the following:

```
-- Raise used by Express code, not intended for you to override.
raise Error(originator=self,
    error=Error.GEN_UNIMPLEMENTED).GetException;
```

In the above example, the Error method creates an instance of the Error class.

Internationalizing Express Windows

To internationalize an Express window, edit the customizable class's window in the Window Workshop (remember you can select the class, choose the Component > Customize… command, and then double-click the Window Workshop category). In the Properties dialogs for the window's widgets, enter a message number for each piece of text (you can specify a default message set number for the entire window on the window's Properties dialog). Each message number corresponds to a message in a message catalog that you must supply with the translated text.

Express automatically invokes the ReloadLabelText method on the window just before the window is opened to cause the appropriate message catalog to be used and the translated message text to be displayed. If you wish to allow users to change the language a window displays while the window is displayed, then you must invoke the ReloadLabelText method in the appropriate place in your code. Express will not translate message text if no message catalog exists for the current application—it will use the default text entered in the Window Workshop. See the *iPlanet UDS Programming Guide* for information about the ReloadLabelText method.

The message catalog you build to translate the messages described above must also contain the message text for built-in components on the generated window (components defined in the ExpressWindows project). These messages are provided in a file in the Express distribution. When you compile your message catalog, you must also compile in the contents of message text file:

$FORTE_ROOT\userapp\express\cl#\message\winen_us.msg

This file contains message text in the message set 60100. The built-in Express messages can be in your message catalog and will not conflict with any message numbers you create, because message sets above 60000 are reserved for iPlanet UDS (message set 60100 is used for Express).

Whether a user-defined message catalog exists or not, Express will use its message catalog to access status and error message text, as it did in Release 1 (the US English version of this file is in: `$FORTE_ROOT\install\nls\exmsg\en_us.cat`).

For details about message catalogs and how to compile them, see *iPlanet UDS Programming Guide*.

Customization Techniques: Window Classes

Express provides a set of window commands and behavior as a result of your specifications in the Application Model Workshop. However, there may be times when you need to modify the generated behavior.

This section shows examples and describes coding techniques you should employ when customizing window classes. In these examples, class names, window names, and attribute names are used for illustration. You will need to change these to be appropriate for your window class. Also, the "method" statements below should not be typed into the Method Workshop, they are provided for your information.

Setting Widget State

Window widgets can be divided into three types:

- built-in widgets defined in classes in the ExpressWindows project: menu items, buttons, and grid fields

- widgets generated onto the window, based on setting in the Application Model Workshop

- widgets added to the window as a window customization

Enabling, disabling, and other widget-state manipulations of these three types are discussed in the following sections.

Built-in Widget States

The generated window classes contain a CommandMgr attribute, of type CommandMgr, which controls the various built-in buttons and menu items that can appear on an Express-generated window. Use this class to manipulate the built-in menu items, buttons, and so on, on your generated windows. If a command appears both as a menu item and as a button on the toolbar, then the CommandMgr class will automatically take care of them both. See the discussion in "CommandMgr" on page 53 in Chapter 1, "Express Application Architecture."

Also note that the built-in menu items and controls present on your windows are affected by the window and link properties you set in the Application Model Workshop. If you want to remove a menu item or control and cannot do so in the Application Model Workshop, then you may delete it from the generated window in the Window Workshop. You may add your own built-in commands to all your windows as a global customization (see "Global Customization" on page 153), or to a single window in the Window Workshop. However, note that Express will not try to manage the state of your button. For example, the CommandMgr class will not know about your menu items and controls; it will not disable them when the user is in Search mode, and so on), so you must do this on your own.

The CommandMgr class defines constants that you should use to refer to the built-in menu items and buttons you want to modify. The constants are named with a prefix of C_RS for Result Set commands, and C_WC for Window commands.

The following examples illustrate using the CommandMgr class to check built-in component states:

```
CommandMgr.SetState(
   command = CommandMgr.C_WC_PREFERENCES,
   state = FALSE );
```

```
CommandMgr.HideCommand(
           command = CommandMgr.C_RS_INSERT );
```

```
CommandMgr.ShowCommand(
           command = CommandMgr.C_RS_INSERT );
```

Note that Express enables and disables commands as the user switches modes, so you may need to hide a command in multiple overridden methods if you want it to stay hidden. If you want a command to go away and never come back, use the RemoveCommand method in the CommandSetDesc class.

Customized Widget States

The state of user-added (customized) widgets will not be modified by Express, so you can control these widget states whenever and however you want. You can, for example, override the PostOpenInit method and set the state of your widgets there. You may set the state directly, or use the SetWidgetState method shown above—the advantage of SetWidgetState method is that it will also set the widget color to match the Express generated fields in the same state.

When adding window attributes, do not forget to use "new()" to create them in the window's Init method, and of course to call super.Init() there.

Finding the Focus Field

Use the GetFocusField method to access the field that currently has the input focus. This method can be called from any window in an Express window (for example, even from a nested or folder window). GetFocusField returns the same result whether it is called from the outermost window or from a nested or folder window.

```
theFocusField : FieldWidget = GetFocusField();
if theFocusField <> NIL then
 -- focus field found
end if;
```

Determining If a User Has Changed Data

To determine if a user has changed data displayed in a window, use the IsResultSetModified attribute on the window. For example:

```
if self.IsResultSetModified = TRUE then
    -- user has changed data in this window
end if;
```

Project: CustomClientTutorialAppWindows
• **Class:** CUSTOMERORDERWindow • **Method:** Save

Creating a New Instance
of a Business Class in a Window

You can customize your application to insert data into the result set, but rather than using "new()" to create an object, you can use the NewObject method to create an object that is initialized for the particular type of BusinessClass.

For example, if window CUSTOMERORDERWindow is based on class CUSTOMERORDERClass, then create a new instance of CUSTOMERORDERClass using the following code in a window method:

```
record : BusinessClass = NewObject();
```

The new CUSTOMERORDERClass object would have references to the appropriate objects. See "Providing Automatic Append on Insert in an Array Window" on page 147 for another example of the NewObject method.

Working with an OutlineField

Express generates four additional methods when a window contains an outline field (you create an outline window in the Application Model Workshop by setting the Layout of Fields property to Outline in a Business Class Window's properties dialog):

• GetCurrentDisplayNode

• GetCurrentIndexNode

• SetCurrentDisplayNode

• SetCurrentIndexNode

These methods are provided to let you access the data currently being displayed in an outline field. See the iPlanet UDS online Help for information about these methods.

The following sections illustrate typical customizations you can make using these methods.

Getting the Currently Selected Display Node

In an Outline field that contains LINEITEMClass records, get the currently selected node using the GetCurrentDisplayNode method. For example:

```
curINode : LINEITEMNode =
            GetCurrentDisplayNode();
```

Getting the Currently Selected Outline Index Node

To access the object that has pointers to the current record and the current display node, use the GetCurrentIndexNode method. For example:

```
curINode : OutlineIndexNode =
            GetCurrentIndexNode();
```

Replacing the Currently Selected Display Node

In an Outline field that contains LINEITEMClass records, replace the currently selected node with a new one using the SetCurrentDisplayNode method. For example:

```
newNode : LINEITEMNode = new();
    ...
SetCurrentDisplayNode(node = newNode);
```

Replacing the Currently Selected Outline Index Node

To replace the object that has pointers to the current record and the current display node, use the SetCurrentIndexNode method. For example:

```
newINode : OutLineIndexNode = new();
    ...
SetCurrentIndexNode(node = newINode);
```

Using a Drilldown Link to a Callout to Close an Outline Window

You can allow the user to close an outline window by double-clicking on a row. This is a convenient way for the user to select data in an outline window and return it to the calling window.

➤ **To use a drilldown link to a callout from an outline window**

1. In your application model, create a custom callout called CloseWin.

2. Enter "FinishUp();" in the TOOL Code field.

3. Create a drilldown link from the outline window to this callout.

4. Regenerate your application.

The callout code will be placed in the DrillDownLink method on the outline window's base window, as shown below:

```
PARTBaseWindow.DrillDownLink(data:Array of BusinessClass,
  query:BusinessQuery, appData:Object): Array of BusinessClass
FinishUp();
```

Project: CustomClientTutorialAppWindows • **Class:** PartBaseWindow
• **Method:** DrillDownLink

Getting Information Passed by the Parent Window

This section describes convenient methods you can use to access information passed to a window by another (parent) window.

Getting Application-Specific Data

Use the GetAppData method to get the application-specific information passed to a window by its calling window. Given a calling window that has a "Before Command Link" customization, in which it passes an object of type CUSTOMERClass to the receiving window as the application data, the receiving window can access that object using this method:

```
cust : CUSTOMERClass = CUSTOMERClass(GetAppData());
if cust <> NIL then
 -- manipulate CUSTOMERClass object
end if;
```

Getting the Initial Result Set

Use the GetInitialRecords method to get the initial result set passed to a window by its calling window. For example, an initial result set, rather than a query, will be passed if two windows both display records of type PARTClass, and in the Application Model, the Display property of the link between the two windows is set to "All PART records displayed in the calling window." The called window can access that initial result set using GetInitialRecords, for example:

```
initialResultSet : Array of PARTClass =
    (Array of PARTClass)(GetInitialRecords());
if initialResultSet <> NIL then
 -- initial resultset was passed
end if;
```

Getting the Initial Query

Use the GetInitialSearch method to get the initial query passed to a window by its calling window. For example, a query rather than a result set will be passed to a window if the called window displays data associated with the current row in the calling window. The called window can access that initial query using GetInitialSearch, for example:

```
initialSearch : BusinessQuery = GetInitialSearch();
if initialSearch <> NIL then
 -- initial query was passed
end if;
```

Get Parent Current Record

Use the GetParentCurrentRecord method to access the current record in the parent (calling) window. For example:

```
parentRec : BusinessClass = GetParentCurrentRecord();
if parentRec <> NIL then
 -- parent record is available
end if;
```

Get Parent Window

Use the GetParentWindow method to access the calling window's UserWindow object. For example:

```
parentWin : ExpressClassWindow = GetParentWindow();
if parentWin <> NIL then
 -- parent window accessed
end if;
```

Customization Techniques: Business Rules on the Client

Business rules are special data requirements that you want your application to enforce automatically, or particular actions the application must perform based on the state of the data. This section describes how to provide several types of data validation, as well as calculate derived fields.

Window Validations

In general, client validation logic must be custom coded for each window. However, with domains, most of the work can be coded one time only in each domain by creating a Validate (or any other name of your choosing) method there and invoking that method when a field validation is required (see the online example "Validating Fields" listed in "Application Model Customization Examples" on page 94). For more information about working with domains, see the online example "Validating Domains" listed in "Application Model Customization Examples" on page 94.

Field Validation Sequence of Events

The generated BaseWindow defines a DataEvents event handler, which handles the AfterValueChange event for every editable field in the window (see the iPlanet UDS online Help for information about the DataEvents event handler). Inside this AfterValueChange block, the following occurs inside a HandleValueChange method (in order—note that the SetValue method on the mapped field attribute has already occurred before this sequence begins):

- a ValidateField method is invoked with a parameter to indicate the current field Id (see "BusinessClass Attribute IDs (ATTR_)" on page 114). An empty version of the ValidateField method is implemented in the ExpressClassWindow class; it is intended that you override this method for each window, to contain a block for each field you want to validate. Raise an exception if a validation error occurs.

- a LogAttr method is invoked to note that the current *business_class*Class attribute value has changed. See "Using the LogAttr Method" on page 115.

- a SetValue method is invoked on the current field's corresponding data item in the current row of DisplayedResultSet. This sets the value in the result set to be the same as the mapped data value in the window. This step is not done for array fields because the array is already mapped directly to the data. The window attribute IsResultSetModified is set to TRUE.

- a FieldValueChanged method is invoked identically to the ValidateField method described above. FieldValueChanged is also implemented as a hook in the ExpressClassWindow class, and it is intended that you override it in your window classes. At this point in the processing, you know that your field value is valid. Thus, you can override the FieldValueChanged method to calculate values of other fields. See the online example "Calculating a Derived Field" listed in "Application Model Customization Examples" on page 94.

Other Business Rules

For other examples that add business rules to the client, see the following examples in the table in "Application Model Customization Examples" on page 94:

- Checking That Data is Entered in Mandatory Fields (Window Data/Validate Records)

- Posting an Event on a Changed Field Value (Window Data/Field Value Changed)

- Calculating a Derived Field From Nested Window Data (Window Data/Field Value Changed)

Customization Techniques: Result Sets

This section discusses a variety of customizations that allow you to change field values and add or remove rows from a result set.

Business Class Record Status

Records in the query result set contain InstanceStatus attributes whose values indicate what changes have been made to a record since it was loaded from the database. These status values are used to determine which queries to run on behalf of a record at Save time. The following constants, defined in the BusinessClass class, describe each numeric value for InstanceStatus:

Constant	InstanceStatus Value	Meaning
ST_READONLY	2	Record is read only and cannot be modified. No Update/Insert/Delete queries will be run on behalf of this record.
ST_READWRITE	4	Record was loaded from the database and is updateable, but has not been changed by the user. Once changed, state will become ST_UPDATE.
ST_UPDATE	8	Record has been modified since being selected from database. Update statement will be run.
ST_INSERT	16	Record is newly created, contains values entered by the user, and is not yet in database. Insert statement will be run.
ST_DELETE	32	Record has been deleted (not yet deleted from database). Delete statement will be run.
ST_EMPTY	1	Empty record to be filled in by user—user has not yet typed any values into the record. (When values are entered, state will become ST_INSERT).

BusinessClass Attribute IDs (ATTR_)

Many methods require an integer parameter to indicate the attribute ID of the business class. The attribute IDs for each business class attribute are generated as constants in class *business_class*BaseQuery. The constants are named the same as the *business_class*Class.attribute name, but with the prefix "ATTR_". For example, the attribute CUSTOMERClass.CUSTOMERNUMBER has a corresponding constant ATTR_CUSTOMERNUMBER defined in CUSTOMERBaseQuery.

Note that when two business classes contain an identically named field (often it is a database join field), the values of the generated ATTR_ constants in the two BaseQuery classes will not necessarily be identical. For example, do not assume the following have the same value:

CUSTOMERBaseQuery.ATTR_CUSTOMERNUMBER

CUSTOMERORDERBaseQuery.ATTR_CUSTOMERNUMBER

Since these constants are inherited, the examples will refer to them through the generated class. For example, the examples will refer to CUSTOMERQuery.ATTR_NAME, rather than CUSTOMERBaseQuery.ATTR_NAME, where it is defined.

Getting and Setting the Value of a Displayed Field

There are cases where the displayed value for a field is stored in both an attribute of the window class (as is done in "standard" iPlanet UDS windows), and in a row of the DisplayedResultSet attribute. To assign a new value to the field corresponding to an attribute in *business_class*Class, use the PlaceValueInDisplayedField method (see the iPlanet UDS online Help for details). The PlaceValueInDisplayedField method is the recommended way to change the displayed field value because it automatically performs a number of steps, such as invoking the LogAttr method. The following example illustrates:

Code Example 2-1   CustomClient4App Example

```
myAddr : TextData = new(value = '1800 Harrison St');
-- PlaceValueInDisplayedField will update the
-- DisplayedResultSet and display the new value in the field.
-- Note that the field parameter is case sensitive.
```

Code Example 2-1   CustomClient4App Example

```
PlaceValueInDisplayedField(field = 'ADDRESS', data = myAddr);
```

Project: CustomClient4AppWindows • **Class:** CustomerWindow
• **EventHandler:** CustomEvents

Accessing the Value of a Field in Search Mode

When a generated Express window is in Search mode, data fields will have a
widget state of FS_QUERY. In this state, field values are not copied to their
mapped attribute. Therefore, to access the value of a field while in search mode,
you should access the field's TextValue attribute, for example:
<CUSTOMERNUMBER>.TextValue.

Changing the Value of an Attribute

In cases where an attribute of a BusinessClass is not displayed, you cannot use the
PlaceValueInDisplayedField method. You will need to modify the attribute
directly.

Express applications track which attributes of each BusinessClass object have been
modified, so the correct queries can be issued later when the user presses Save. If
you need to modify an attribute value as a customization and want it logged so the
new value will appear in a subsequent Update or Insert statement for that object,
then you need to use the LogAttr method (see below), in addition to doing a
SetValue on the attribute. For example, to modify the NAME attribute of a
window's current record (CUSTOMERClass), do the following:

```
BusinessClient.LogAttr( GetCurrentRecord(),
    CUSTOMERQuery.ATTR_NAME );
GetCurrentRecord().NAME.SetValue( 'Paul' );
```

Using the LogAttr Method

If you want to make changes to the result set and want those changes to be part of
subsequent update or insert queries, then you will need to use the LogAttr method.

Every BusinessClass object references a BusinessQuery object (reference may be NIL). LogAttr saves information about modifications to a BusinessClass object in the BusinessQuery object referenced by the BusinessClass object (in BusinessClass.UpdateQuery.Values attribute). If LogAttr has been run on a BusinessClass, then this will cause an Update or Insert SQL statement to be run for that BusinessClass object at save time (Update or Insert is determined based on the State attribute of the BusinessClass object, as described in "Business Class Record Status" on page 113). Running LogAttr on an attribute causes that attribute to be added to the query.

The following examples illustrate using the LogAttr method (BusinessClient is an attribute on the window class):

Code Example 2-2   Mark that attribute CUSTOMERNUMBER in a BusinessClass object is about to be updated

```
BusinessClient.LogAttr(source = myCust, attr =
  CUSTOMERORDERQuery.ATTR_CUSTOMERNUMBER);
```

Note that LogAttr should be run on an attribute *before* the value is changed, because the optimistic concurrency option requires a before-image of the object at update time, as does the Revert feature (see Chapter 1, "Express Application Architecture" for information on workshop properties and generated classes).

You can either use LogAttr before changing the attribute value in each object, or you can use LogAttr on the entire object without specifying an attribute, as shown below. If you do not specify an attribute, then the entire class is logged, regardless of which attribute has changed. This ensures that any new values are sent to the database at save time.

Code Example 2-3   Mark that some attribute in a BusinessClass object has been updated

```
BusinessClient.LogAttr(source = myCust);
```

After that, you can use LogAttr either before or after the attribute value changes. If you use LogAttr on an attribute that does not change, this will generate an update clause where the column is updated with the same value.

Checking Query Information on a BusinessClass Object

There are several operations that can be run on a BusinessClass object to check whether the LogAttr method has been run (the object myCust below is of type CustomerOrderClass):

- Check if a BusinessClass object has been updated (has LogAttr been run on the class, or on any attribute of the class) and whether the ADDRESS attribute has been updated:

```
myCust :  CUSTOMERClass = GetCurrentRecord();

if (myCust <> nil) then
  if (myCust.UpdateQuery = NIL) or
      (mycust.UpdateQuery.Values = NIL) then
    task.Part.LogMgr.PutLine(
      'Business class has not been updated.');
  else
    task.Part.LogMgr.PutLine(
      'Business class has been updated.');
    if (myCust.UpdateQuery.GetUpdateAttr(
        attr = CUSTOMERQuery.ATTR_ADDRESS) = NIL) then
      task.Part.LogMgr.PutLine(
      'Address field has not been updated.');
    else
      task.Part.LogMgr.PutLine(
      'Address field has been updated.');
    end if;
  end if;
end if;
```

Example App: CustomClient4App • **Project:** CustomClient4AppWindows
• **Class:** CustomerWindow • **Eventhandler:** CustomEvents

- Reset a BusinessClass object so it behaves as if it was just selected from the database. This will reset the status and "forget" any LogAttr information:

```
myCust : CUSTOMERClass = GetCurrentRecord();
if (myCust <> nil) then
  myCust.Reset();
end if;
```

Example: CustomClient4App • **Project:** CustomClient4AppWindows
• **Class:** CustomerWindow • **Eventhandler:** CustomEvents

Note that Reset is not the same as RevertToSaved. The Reset method does not change the value of attributes in the DisplayedResultSet, but simply changes the InstanceStatus and UpdateQuery attributes on the BusinessClass object.

Looping Through a Displayed Result Set

You can loop through the displayed result set as you would any array. The row type will be the BusinessClass object (such as CUSTOMERClass). There will be attributes corresponding to the attribute names in your business object. The following code illustrates looping through a displayed result set:

```
for drs in DisplayedResultSet do
    if drs.CUSTOMERNUMBER.Value > <somevalue> then
        -- do some processing...
    end if;
end for;
```

Example App: CustomClient4App • **Project:** CustomClient4AppWindows
• **Class:** CustomerWindow • **Eventhandler:** CustomEvents

Using Displayed Result Sets with Outline Fields

For windows that have their Layout of Fields property set to Outline, the DisplayedResultSet attribute will be defined as "Array of OutlineIndexNode" (each OutlineIndexNode contains a ResultSetNode attribute, of type BusinessClass, and an OutlineNode attribute, of type DisplayNode). This differs from other field layouts (form and array), where DisplayedResultSet is defined as "Array of *business_class*Class" (for example, array of CUSTOMERClass). The Express methods described in this section, such as GetCurrentRecord, continue to work correctly for outline fields (see the iPlanet UDS online Help for information about the GetCurrentRecord method). However, you will have to modify direct access to the DisplayedResultSet window attribute. For example, the above example code would have to be modified to the following if the window's field layout is Outline:

```
for drs in DisplayedResultSet do
  if (CUSTOMERClass(drs.ResultSetNode).CUSTOMERNUMBER.Value
    < 100) then
      -- do some processing...
  end if;
```

```
end for;
```

Example App: CustomClient6App • **Project:** CustomClient6AppWindows
• **Class:** CustomerWindow • **Eventhandler:** CustomEvents

Removing Rows from a Result Set

To remove the currently displayed row from the displayed result set and mark it for deletion from the database, you would use the DeleteRecordFromResultSet method, as shown:

```
-- Remove record and mark it for deletion.
DeleteRecordFromResultSet(confirm = FALSE);
```

Example App: CustomClient6App • **Project:** CustomClient6AppWindows
• **Class:** CustomerWindow • **Eventhandler:** CustomEvents

Displaying a Row in a Result Set

To display a specific row from the result set, you would use the SelectRecord method, as shown below:

```
row : integerData = new(value = 2);
SelectRecord(theIndex = row);
```

Example App: CustomClient6App • **Project:** CustomClient6AppWindows
• **Class:** CustomerWindow • **Eventhandler:** CustomEvents

Accessing a Nested Result Set

There are two attributes that allow you to access the result set in a nested window. For example, in a CustomerOrder window with a nested LineItem window, the following refer to data of type Array of LINEITEMClass:

• GetCurrentRecord().LINEITEM (this corresponds to the nested ResultSet, which includes deleted records)

• LINEITEMLink1Nested.DisplayedResultSet

Note that LINEITEMLink1Nested is of type LINEITEMWindow, which is the window class of the nested window. Of these two, GetCurrentRecord().LINEITEM is preferred because it is more direct and is based off the current record. So, to loop through the line items for the current CustomerOrder, you would use the following code:

```
for li in GetCurrentRecord().LINEITEM do
  if (li.InstanceStatus <> BusinessClass.ST_DELETE) then
    if li.PARTNUMBER.Value = 90001 then
        -- do something to this lineitem.
    end if;
  end if;
end for;
```

Example App: CustomClient5App • **Project:** CustomClient5AppWindows
• **Class:** CustomerOrderWindow • **Eventhandler:** CustomEvents

See the iPlanet UDS online Help for more information about these generated attributes.

Customization Techniques: Queries

Express does not generate pre-defined queries. Select queries are built at runtime by the window class. The target list in any Select query is based on the fields displayed on the generated window. This information is generated into the method *business_class*BaseWindow.GetRecordTemplate, and the Select WHERE clause is based on the QBE-style entries made by the end user while the window is in Search mode. For updates and inserts, the target list is based on which object attributes are marked as changed (see "Using the LogAttr Method" on page 115). For updates and deletes, the WHERE clause is based on the keys specified for the class in the business model.

You can use the same techniques used on the Express windows classes to build queries to augment the generated queries. This can be done in either the client or business service.

Express queries are always executed by the business service. As with any database application, query performance is much better if performed on the server, rather than the client.

Queries are encapsulated into an object of class *business_class*Query (such as LINEITEMQuery). The classes *business_class*Class and *business_class*Query reference each other (BusinessClass.UpdateQuery, BusinessQuery.OriginalClass, and BusinessQuery.UpdatedClass). See the class diagram in for details.

Modifying Generated Queries

The queries constructed by Express can be modified by adding columns to the target list or modifying the WHERE clause. Queries can also be tested to see which business class they are for, and which business class attributes participate in the query. The following example illustrates.

The variable "myQuery" in the example below refers to an object of type BusinessQuery (superclass of *business_class*Query).

Code Example 2-4   Check if a Query is for CUSTOMERClass

```
if myQuery.IsA(CUSTOMERQuery) then
    -- query for CUSTOMERClass
end if;
```

Code Example 2-5   Check if the target list of a query contains attribute ADDRESS

```
if myQuery.HasAttr(attr =
   CUSTOMERQuery.ATTR_ADDRESS)
then
    -- target list contains ADDRESS.
end if;
```

Example App: CustomQueryApp • **Project:** CustomQueryAppWindows
• **Class:** CustomerWindow • **Method:** GetRecordTemplate

Select Queries

Add columns to the select query target list using the *business_class*Query.AddAttr method, where the parameter specifies the attribute to add:

Code Example 2-6   Adding columns to the select query target list

```
myQuery.AddAttr(
  attr = CUSTOMERORDERQuery.ATTR_CUSTOMERNUMBER);
```

Execute a select query using the Select method on the BusinessClient class (for example, TutorialClient). This method executes the query and returns an array of BusinessClass as the query results:

```
tempQuery : PARTQuery = PARTQuery(
    RecordTemplate.GetQuery( LINEITEMQuery.ATTR_PART ) );
returnSet : Array of BusinessClass;
...
if BusinessClient.TransActive() = TRUE then
  returnSet = BusinessClient.Select( query = tempQuery,
          TransactionMode = ConcurrencyMgr.TR_CONTINUE );
else
  returnSet = BusinessClient.Select( query = tempQuery,
          TransactionMode = ConcurrencyMgr.TR_START );
end if;
tempQuery.Reset();
```

Example App: CustomClientTutorialApp
- **Project:** CustomClientTutorialAppWindows • **Class:** LineItemWindow
- **Method:** ValidateField

WHERE Clause Parameters

Add constraints to the query WHERE clause using the *business_class*Query.AddConstraint method, in whose parameters you specify the business class attribute for which to add a constraint, the value of the constraint, and the operator. Use the following constants to refer to the operator:

- ConstraintOperation.OP_EQ for "="

- ConstraintOperation.OP_GT for ">"

- ConstraintOperation.OP_LT for "<"

```
tempQuery.AddConstraint( attr = PARTQuery.ATTR_PARTNUMBER,
               operation = ConstraintOperation.OP_EQ,
               value = TheCurrentRecord.PARTNUMBER );
```

Example App: CustomClientTutorialApp
- **Project:** CustomClientTutorialAppWindows• **Class:** LineItemWindow
- **Method:** ValidateField

Constructing a New Query

Express generates queries for the types of operations needed in the generated windows based on settings in the Business Model and Application Model Workshops. However, there may be times when you need to issue queries that are not represented in your application model. You can do this by either:

- adding an SQL query to your application's TOOL code and creating a new database service on which to execute the query

- building up an Express BusinessQuery object and then executing it in the business service

The second of these two techniques requires that the table you are querying be known in your Business Model; the first technique does not. This section shows the second option.

NOTE   You cannot construct and execute queries on a component class—that is, a class that has an aggregate relationship with another class. Any business class on which you construct a query must have a corresponding *business_class*Mgr class (for example, CustomerMgr). However, a component class does not have an associated *business_class*Mgr class, because it is managed by its aggregate class. Thus, for example, you cannot construct a query on the LineItem class as it is used in the Tutorial business model.

In this example, you will issue queries against the Customer table in a window that is based on CUSTOMERClass.

Select Query

In your Customer window, you will select all the customers with a customer number that is equal to or greater than 2. You will execute the following query, which will return an array of CUSTOMERClass objects:

```
select CUSTOMERNUMBER, NAME, ADDRESS, PHONE
from CUSTOMER
where CUSTOMERNUMBER >= 2
```

To do this, create a method in the CUSTOMERWindow with the following code:

```
-- Create a new query object and add the attribute list.
cq  : CUSTOMERQuery = new;
cq.AddAttr( attr = CUSTOMERQuery.ATTR_CUSTOMERNUMBER);
cq.AddAttr( attr = CUSTOMERQuery.ATTR_NAME);
cq.AddAttr( attr = CUSTOMERQuery.ATTR_ADDRESS);
cq.AddAttr( attr = CUSTOMERQuery.ATTR_PHONE);

-- Add a constraint to the query.
cust : IntegerData = new(value = 2);
cq.AddConstraint(
  attr = CUSTOMERQuery.ATTR_CUSTOMERNUMBER,
  value = cust,
  operation = ConstraintOperation.OP_GE);

-- Execute the Select query.
customer_array : array of CUSTOMERClass =
  (array of CUSTOMERClass)(BusinessClient.Select(
    query = cq,
    transactionMode = ConcurrencyMgr.TR_START));
```

Complex Select Query

This method illustrates how to construct a multi-table join query. It does not rely on any data from the CUSTOMERORDERWindow class. It could be constructed in any window.

We construct a Select query that selects a set of CustomerOrder table rows and matching data from the Customer and LineItem tables. In addition, each LineItem row must contain matching data from the Part table.

Because of optimizations made by the Express services partition, the query we execute will be run as two queries: one to select all the CustomerOrder rows (doing a join to Customer), then a second query to select all the LineItem rows to match all the selected CustomerOrder rows (and doing a join between LineItem and Part). The Express services partition then breaks up the set of selected LineItem rows and associates each group with its corresponding CustomerOrder row. The result set returned to the client looks like each CustomerOrder row had a separate query run to select its associated LineItem rows. However, all the results were obtained with only two queries.

Code Example 2-7   Complex Select Query Example

```
method CUSTOMERORDERWindow.ComposeQueries
begin

-- Define BusinessQuery objects.
customerOrderQ : CUSTOMERORDERQuery = new();
customerQ : CUSTOMERQuery = new();
lineItemQ : LINEITEMQuery = new();
partQ : PARTQUERY = new();

-- Set fields to retrieve from CustomerOrder table.
customerOrderQ.AddAttr(
  attr = CUSTOMERORDERQuery.ATTR_CUSTOMERNUMBER);
customerOrderQ.AddAttr(
  attr = CUSTOMERORDERQuery.ATTR_ORDERNUMBER);
customerOrderQ.AddAttr(
  attr = CUSTOMERORDERQuery.ATTR_REQUESTEDDATE);

-- Set query restriction: OrderNumber >= 1000
customerOrderQ.AddConstraint(
  attr = CUSTOMERORDERQuery.ATTR_ORDERNUMBER,
  value = IntegerData(value = 1000),
  operation = ConstraintOperation.OP_GE);

-- Set fields to retrieve from Customer table.
customerQ.AddAttr(attr = CUSTOMERQuery.ATTR_CUSTOMERNUMBER);
customerQ.AddAttr(attr = CUSTOMERQuery.ATTR_ADDRESS);

-- Request that the Customer attribute of CUSTOMERORDERClass
-- be filled in as specified by CUSTOMERQ. This will be
-- executed as a CustomerOrder-to-Customer join.
customerOrderQ.AddAttr(
  attr = CUSTOMERORDERQuery.ATTR_CUSTOMER,
  query = customerQ);

-- Set fields to retrieve from LineItem table.
-- 'ATTR_SIMPLE' means all LineItem non-reference
-- attributes.
lineItemQ.AddAttr(attr = LINEITEMQuery.ATTR_SIMPLE);
```

Code Example 2-7   Complex Select Query Example *(Continued)*

```
-- Request that the LineItem attribute of
-- CUSTOMERORDERClass be specified by LineItemQ.
-- This will be executed as a CustomerOrder-to-LineItem
-- join.
customerOrderQ.AddAttr(
  attr = CUSTOMERORDERQuery.ATTR_LINEITEM,
  query = lineItemQ);

-- Set fields to retrieve from Part table.
-- 'SIMPLE_ATTR' means all Part non-reference
-- attributes.
partQ.AddAttr(attr = PARTQuery.ATTR_SIMPLE);

-- Request that the Part attribute of LINEITEMClass
-- be filled in a specified by partQ. This will be
-- executed as a LineItem-to-Part join.
lineItemQ.AddAttr(
  attr = LINEITEMQuery.ATTR_PART,
  query = partQ);

-- Execute the Select query.
orders : array of CUSTOMERORDERClass =
  (array of CUSTOMERORDERClass)(BusinessClient.Select(
  query = CustomerOrderQ,
  transactionMode = ConcurrencyMgr.TR_SINGLETON));

-- Write the data returned by the query to the logger.
bc : BusinessClass = new();
task.Part.LogMgr.PutLine(bc.FillString(
  source = orders));
end method;
```

Example App: CustomQuery3App • **Project:** CustomQuery3AppWindows
• **Class:** CustomerOrderWindow • **Method:** ComposeQueries

Update Query

The example in this section manipulates the objects in the customer_array selected above. The example then invokes the BusinessClient.Update method, which issues database queries on each row according to its status.

Note that the initial status of each *business_class*Class object is significant, because
the status is used by the LogAttr method to set additional state information and
create other objects needed so the appropriate SQL query will be executed later by
the Update method. The rows in the customer_array begin with status
ST_READWRITE, which was given them by the BusinessClient.Select method
when the objects were selected. See "Business Class Record Status" on page 113 for
more information about the BusinessClass object status.

```
-- Find the largest CUSTOMERNUMBER
max_cust : integer = 0 ;
for c in customer_array do
  if c.CUSTOMERNUMBER.value > max_cust then
    max_cust = c.CUSTOMERNUMBER.value;
  end if;
end for;

-- Add three new CUSTOMERClass objects to array.
i : integer = 0;
while i  <  3 do
  c: CUSTOMERClass = new;
  c.CUSTOMERNUMBER = new;
  c.NAME = new;
  c.ADDRESS = new;
  c.PHONE = new;

  max_cust = max_cust + 1;
  text:TextData = new;
  c.CUSTOMERNUMBER.SetValue(max_cust);

  text.SetValue('CUSTOMER ');
  text.Concat(max_cust);
  c.NAME.SetValue(text);

  text.SetValue('ADDRESS FOR CUSTOMER ');
  text.Concat(max_cust);
  c.ADDRESS.SetValue(text);

  text.SetValue('510-777-2222');
  c.PHONE.SetValue(text);

--This is a shorthand way of changing all datavalue objects.
  BusinessClient.LogAttr(source = c, attr =
BusinessQuery.ATTR_SIMPLE);
  customer_array.AppendRow(object = c);
  i = i + 1;
end while;

-- Mark the last item in the CUSTOMERClass array as Deleted.
BusinessClient.Delete( source =
customer_array[customer_array.items]);
```

```
-- Execute the Save
BusinessClient.Update(source = customer_array,
  transactionMode = ConcurrencyMgr.TR_END);
```

Examining the Generated SQL

You can access the text for each generated query by overriding *business_class*Mgr methods for each businessClass you are interested in:

- override the ExecuteSql or SQLUpdate/SQLInsert/SQLDelete methods for Update/Insert/Delete queries
- override the SQLSelect method for select queries

The "SQLText" parameter to these methods is the text of the query to be run, but with parameters to the query (such as values from the BusinessClass object or values entered by the user)— appearing as "?". These are SQL placeholders. The SQLData parameter gives the values that will be substituted for the placeholders at runtime. You can modify the query text (parse it, add new clauses, and so on) in a method that overrides one of these methods. When you pass the modified query text to the super.*method*, the modified query text will be executed.

NOTE	You can cause the query text to be printed to the Server Partition's log file by setting the logger flag (-fl) to "trc:ex:1-2:255." See the *iPlanet UDS System Management Guide* for information about setting logger flags.

Using TOOL SQL Statements

It is preferable for efficiency reasons to use the queries Express constructs for you whenever possible. However, if needed, you may enter TOOL SQL statements directly in your generated Express Window customization code (just like you would do in a non-Express window). Those SQL statements may also reference the generated DBSession service object (for example, TutorialDBService) in the query's "on session" clause, or you may create a new DBSession service object and reference that. Note that you must first add "GenericDBMS" as a supplier plan to your generated *application_model*Windows project if you plan to create your own DBSession service object in it.

Complex Examples

The following examples offer more complex customizations than are provided in the Customization Manager Help for individual items, or in the previous sections of this chapter. Most of these examples are also available online through the Customization Manager Help, as described in "Complex Examples" on page 97. The examples are:

Using an Express Window as a Login Window

If you plan to use an Express window as your logon window, you should have a table with user information in your database, as well as a business class (User, for example) in your business model based on that table. Your Express login window will be based on this business class.

➤ **To create an application that uses an Express window as its login window**

1. In the Application Model Workshop, create a window based on the User business class.

2. Use the Attribute List to specify that the relevant fields be displayed. Typically, these would be USERNAME and PASSWORD fields.

3. In the Business Class Window Properties dialog for the login window, set the following properties:

Property	Setting
Layout of Fields	Form
Default Interface	Menu and Toolbar

Property	Setting
Command Set	Search

4. Add a command link between the login window and your first "main" window.

5. In the command link's Link Properties dialog, set the following properties:

Property	Setting
Label	'Log on'
Display	No Records
Read Status	Read/Write
Mode Status	Modeless

6. Generate the windows.

7. Open the generated login window in the Window Workshop.

8. Delete the toolbar by selecting a widget in the toolbar, and pressing Ctrl-Click to select the parent until you get the "ToolBarGrid". Once you have selected ToolBarGrid, delete it.

9. In the Menu Workshop, remove the Result Set and Edit menu items from the menu.

10. Override methods in your login window to verify the user is valid and to close down the login window to save memory. These overrides are described below.

Verifying Logon Information

The following method comes from the CustomClient2 example, which uses an Express window as its login window. In this example, a login window is invoked before users can proceed to the Customer window, and the user's name and password are verified in the database.

➤ **To verify a user's login ID and password**

1. Create a CommandLinkTo*link_name*Link1 method in the window, overriding the identically named method in the superclass. This method is invoked when the user presses the CUSTOMERLink1BC button (see the iPlanet UDS online Help for more information).

Code Example 2-8 Override ORDUSERBaseWindow. CommandLinkToCUSTOMERLink1

```
method ORDUSERWindow.CommandLinkToCUSTOMERLink1(
    data  : Array of BusinessClass = NIL,
    query : BusinessQuery = NIL,
    appData : Object = NIL)
    : Array of BusinessClass
-- Check for a field being left blank.
if <USERNAME>.TextValue.IsEqual('') or
  <PASSWORD>.TextValue.IsEqual('') then
  self.Window.MessageDialog(
    'Please fill in both fields.', MT_ERROR);
  ClearResultSet();
  return nil;
-- Check for use of wildcards.
elseif (<USERNAME>.TextValue.MoveToChar('%') = TRUE) or
  (<PASSWORD>.TextValue.MoveToChar('%') = TRUE)
  self.Window.MessageDialog('No wildcards allowed.', MT_ERROR);
  ClearResultSet();
  return nil;

-- Check database to make sure it's a valid user.
else
  loginResult:Array of BusinessClass = Search(GetSearchCriteria());

  if (loginResult = nil) or
    (loginResult.Items <> 1) or
    (loginResult[1].InstanceStatus = BusinessClass.ST_EMPTY) then
    self.Window.MessageDialog('Not a valid user.', MT_ERROR);
    ClearResultSet();
    return nil;
  else
    return super.CommandLinkToCUSTOMERLink1(
      data = data, query = query, appData = appData);
  end if;
end if;
```

Example App: CustomClient2App • **Project:** CustomClient2AppWindows
• **Class:** OrdUserWindow • **Method:** CommandLinkToCUSTOMERLink1

➤ **To close down the login window after the user successfully logs in**

1. Create an attribute on your login window of type ExpressContainerWindow.

   This will be your handle to the first "main" window in your application. You will need to keep this around so that you will later be able to exit the application after the "main" window shuts down. In the CustomClient2 example, this attribute is called MainWindow.

2. Override AfterCUSTOMERLink1Open to set up a handle to CUSTOMERWindow and close down your login window:

   Code Example 2-9   Override ORDUSERBaseWindow.
                            AfterCUSTOMERLink1Open

   ```
   method ORDUSERWindow.AfterCUSTOMERLink1Open(
       linkedWindow  : ExpressContainerWindow)

   -- Give the attribute a handle to the Customer Window,
   -- which we'll need later to shut it down.
   MainWindow = linkedWindow;
   -- Now close the login window to save memory.
   self.FinishUp();
   ```

3. Override StartMethod to actually exit the application after your "main" window shuts down:

   Code Example 2-10   Override ExpressContainerWindow. StartMethod

   ```
   method ORDUSERWindow.StartMethod()

   super.StartMethod();

   -- Need to exit after the Customer window shuts down.
   if MainWindow <> nil then
     event case
       when MainWindow.AfterFinishUp do
         exit;

       when MainWindow.AfterCancelWindow do
         exit;
     end event;
   end if;
   ```

   Example App: CustomClient2App • **Project:** CustomClient2AppWindows
   • **Class:** OrdUserWindow • **Method**: StartMethod

Calculating a Derived Field From Nested Window Data

To calculate a derived field from nested window data, you need to create a custom attribute and override several methods. In this example, you will add a field to the window that will maintain a total count of the number of items ordered.

➤ **To create a custom attribute**

1. In the Business Model Workshop, add a new attribute to the CustomerOrder class.

2. Open the attribute's Properties dialog and, in addition to setting other appropriate properties, select the Custom toggle.

3. In the Application Model Workshop, display the custom attribute in the window based on the CustomerOrder business class (custom attributes can be displayed on the window, but are not used in queries).

➤ **To calculate the value for the custom attribute (derived field)**

1. Create a method to loop through the line items and sum the quantity:

Code Example 2-11   Create CustomerOrderWindow. CountQuantity

```
method CUSTOMERORDERWindow.CountQuantity(
          lineitems=Array of LINEITEMClass) : integer
cnt : integer = 0;
if lineitems <> NIL then
  for item in lineitems do
    -- When a displayed row has been deleted, don't count it.
    if item.InstanceStatus <> BusinessClass.ST_DELETE then
        cnt = cnt + item.QUANTITY.IntegerValue;
    end if;
  end for;
end if;
return cnt;
```

Example App: CustomClientTutorialApp
• **Project:** CustomClientTutorialAppWindows
• **Class:** CustomerOrderWindow • **Method:** CountQuantity

2. Override the AfterChildWindowChange and DisplayCurrentRecord methods
to invoke the CountQuantity method. The first method runs when changes are
made to the nested LineItem window, and the second method runs when the
user moves to a new master record (see also the example "Executing a Query
on a Record Select" listed in "Application Model Customization Examples" on
page 94).

Code Example 2-12 Override ExpressClassWindow. AfterChildWindowChange

```
method CUSTOMERORDERWindow.AfterChildWindowChange(
    window : ExpressClassWindow)
-- check which nested window changed.
if window.IsA(LINEITEMWindow) then
    ItemCount.integerValue = CountQuantity(lineitems =
     LINEITEMWindow(window).DisplayedResultSet);
end if;
super.AfterChildWindowChange(window=window);
```

Code Example 2-13 Override CustomerOrderBaseWindow. DisplayCurrentRecord

```
method CUSTOMERORDERWindow.DisplayCurrentRecord()
theCurrentRecord : CUSTOMERORDERClass = GetCurrentRecord();
if (theCurrentRecord <> nil) and
  (theCurrentRecord.InstanceStatus <> BusinessClass.ST_EMPTY) then
  -- Calculate the ItemCount based on adding the Quantity values.
  ItemCount.IntegerValue = CountQuantity(
    lineItems = theCurrentRecord.LINEITEM);
  theCurrentRecord.ItemCount = new;
  theCurrentRecord.ItemCount.SetValue(ItemCount.IntegerValue);
end if;
super.DisplayCurrentRecord();
```

Example App: CustomClientTutorialApp • **Project:**
CustomClientTutorialAppWindows • **Class:** CustomerOrderWindow
• **Method:** DisplayCurrentRecord

Another way to calculate a derived field from nested window data is to modify the
LINEITEMWindow so it posts an event whenever the user changes a Quantity field
value. This will cause the CountQuantity method code in the parent window to
execute only when the Quantity field changes, rather than when anything changes
in the nested window (the AfterChildWindowChange method will execute after
any change in the nested window). To do this, you modify the above example to:

1. Create the CountQuantity method exactly like step one in the previous example.

2. Create a new event called QuantityChanged on the LINEITEMWindow.

3. Override the FieldValueChanged method to detect a change to the Quantity field; when detected, post the QuantityChanged event:

Code Example 2-14   Override ExpressClassWindow. FieldValueChanged

```
method LINEITEMWindow.FieldValueChanged(
        fieldNum : integer, newValue : DataValue)
if fieldNum = LINEITEMQuery.ATTR_QUANTITY then
  post QuantityChanged;
end if;
super.FieldValueChanged(fieldNum=fieldNum,
                        newValue=newValue);
```

Example App: CustomClientTutorialApp
• **Project:** CustomClientTutorialAppWindows
• **Class:** LineItemWindow • **Method:** FieldValueChanged

4. Override the DeleteRecordFromResultSet method to detect when a LineItem row has been deleted; when detected, post the QuantityChanged event:

Code Example 2-15   Override ExpressClassWindow. DeleteRecordFromResultSet

```
method LINEITEMWindow.DeleteRecordFromResultSet(
        confirm : boolean = FALSE) : boolean
-- We need to post an event to update the ItemCount field when
-- a record is deleted.
retVal : boolean;
retVal = super.DeleteRecordFromResultSet(confirm = confirm);
if retVal = TRUE then
  post QuantityChanged;
end if;
return(retVal);
```

Example App: CustomClientTutorialApp
• **Project:** CustomClientTutorialAppWindows
• **Class:** LineItemWindow • **Method:** DeleteRecordFromResultSet

5. Override the DisplayCurrentRecord method in the
CUSTOMERORDERWindow, as described in step two in the previous
example.

6. Create a CustomEvents event handler in the CUSTOMERORDERWindow and
add the following statements to handle the event raised by
LINEITEMWindow:

Code Example 2-16   Create CustomerOrderWindow. CustomEvents

```
when LINEITEMLink2Nested.QuantityChanged do
    ItemCount.IntegerValue = CountQuantity(lineitems =
      GetCurrentRecord().LINEITEM);
```

Example App: CustomClientTutorialApp
• **Project:** CustomClientTutorialAppWindows
• **Class:** LineItemWindow • **Method:** DeleteRecordFromResultSet

To see this technique in context, see the Express example program
CustomClientTutorialApp. Look for the methods described above in the
CustomerOrderWindow and LineItemWindow classes in the
CustomClientTutorialAppWindows project.

Generating Records with Unique Sequence IDs

This example illustrates looping through the displayed result set to assign
sequence numbers to an attribute that is part of a composite key. The Sequence
method would be invoked before saving by overriding the Save method. See
"Validating Data Before a Save," listed in "Application Model Customization
Examples" on page 94. This approach applies only when you have a one-to-many
aggregate association. Otherwise, you could not guarantee you were adding
unique sequence numbers.

In other circumstances, when you do not have a one-to-many aggregate association, you should use a database stored procedure to generate unique sequence numbers.

Code Example 2-17   Override ExpressClassWindow.Save

```
method CUSTOMERORDERWindow.Save()

self.Sequence();
return super.Save();
```

Example App: CustomClient5App • **Project:** CustomClient5AppWindows • **Class:** CustomerOrderWindow • **Method:** Save

This method will assign sequence numbers to rows about to be inserted.

Code Example 2-18   Create Sequence method

```
method CUSTOMERORDERWindow.Sequence()

maxseq : integer = 0;
-- Find the highest existing LINEITEMNUMBER.
for li in GetCurrentRecord().LINEITEM do
  if (li.InstanceStatus = BusinessClass.ST_READWRITE) or
     (li.InstanceStatus = BusinessClass.ST_READONLY) or
     (li.InstanceStatus = BusinessClass.ST_UPDATE) then

    if (maxseq < li.LINEITEMNUMBER.IntegerValue) then
      maxseq = li.LINEITEMNUMBER.IntegerValue;
     end if;
  end if;
end for;

-- Give sequence numbers to rows about to be inserted.
for li in GetCurrentRecord().LINEITEM do
  if (li.InstanceStatus = BusinessClass.ST_INSERT) then
    maxseq = maxseq + 1;
    li.LINEITEMNUMBER.IntegerValue = maxseq;
  end if;
end for;
```

Example App: CustomClient5App • **Project:** CustomClient5AppWindows • **Class:** CustomerOrderWindow • **Method:** Sequence

Synchronizing Data in a Modeless Linked Window

Normally, a modeless linked window does not need to communicate further with its parent window. However, you may want to keep the linked window's data synchronized with the parent. Then if the user changes records in the parent window, a new query will be run in the linked window to automatically refresh its data.

In the following example, suppose there is a modeless link between the CUSTOMER and CUSTOMERORDER windows, and the Link Properties dialog is set to Display: "CUSTOMERORDER record(s) associated with selected CUSTOMER". Also in this Link Properties dialog, the Called Window fields are set to Read Only and Modeless. If the user opens a CUSTOMERORDER window, then you would like its data to refresh automatically when the user changes rows in the (parent) CUSTOMER window. Here are the steps to accomplish this:

1. Add a new attribute to the CUSTOMERWindow of type CUSTOMERORDERWindow (the type of the linked window). Name the attribute CUSTOMERORDERChild. This attribute will be the parent window's "handle" to the linked window.

2. Override the method that is invoked after the user presses the Link button and the linked window is displayed. Assign the method's "window" parameter to the new attribute created above (cast required):

Code Example 2-19   Override CustomerBaseWindow.
AfterCUSTOMERORDERLink1Open

```
method CUSTOMERWindow.AfterCUSTOMERORDERLink1Open(
        window : ExpressContainerWindow)
-- Save reference to linked window
self.CUSTOMERORDERChild = CUSTOMERORDERWindow(window);
super.AfterCUSTOMERORDERLink1Open(linkedWindow = window);
```

Example App: CustomClient3App • **Project:** CustomClient3AppWindows
• **Class:** CustomerWindow • **Method:** AfterCUSTOMERORDERLink1Open

3. Override the method that is invoked when the user selects a new CUSTOMER record (see also the example "Executing a Query on a Record Select," listed in "Application Model Customization Examples" on page 94) and execute a query in the linked CUSTOMERORDER window.

Code Example 2-20   Override CUSTOMERBaseWindow. DisplayCurrentRecord

```
method CUSTOMERWindow.DisplayCurrentRecord()
if  (GetCurrentRecord() <> nil) and
  (GetCurrentRecord().InstanceStatus <> BusinessClass.ST_EMPTY)
    then
   -- moving to new record.
   -- execute query in child window.
   if (self.CUSTOMERORDERChild <> nil) and
   (CUSTOMERORDERChild.Window.IsOpen = TRUE) then
       SearchCriteria : CUSTOMERORDERQuery =
           CUSTOMERORDERQuery(
       CUSTOMERORDERChild.GetRecordTemplate());
     SearchCriteria.AddConstraint(
       attr = CUSTOMERORDERQuery.ATTR_CUSTOMERNUMBER,
       value = GetCurrentRecord().CUSTOMERNUMBER,
       type = IntegerDomain() );
    CUSTOMERORDERChild.SetResultSetFromQuery(
       SearchCriteria );
  end if;
end if;
super.DisplayCurrentRecord();
```

Example App: CustomQueryApp • **Project:** CustomClient3AppWindows
• **Class:** CustomerWindow • **Method:** DisplayCurrentRecord

4. Edit the Class Properties for the linked (child) CUSTOMERORDERWindow and make it Shared.

NOTE   CustomerOrderWindow must have the Shared property set to IsDefault=On. The event loop of the CustomerWindow is running in one task; this event loop is initiating a change on the CustomerOrderWindow that has an event loop running in a different task. To avoid conflicts where two tasks try to modify displayed data simultaneously, use the Shared Object property. This prevents two methods on the CustomerOrderWindow from executing simultaneously. See the *TOOL Reference Guide* for information about the Shared property.

Customizing the Database Mapping of a Business Class

The Business Model Workshop assumes that the attributes in a business class all come from the same database table. This example shows what to do when attributes in a business class come from *different* database tables.

In this example, there are two tables in the database that have a one-to-one join relationship. However, you only want to specify one of the tables as a business class in the Business Model Workshop. The second table will be handled by customizations to the generated BusinessQuery class.

The Part table has a one-to-one relationship with the PartDesc table. The key to table Part is column PartNumber, and the key to PartDesc is column PartNo. Besides the key, PartDesc has one column: a long text field that describes how to construct the part (FullDescrip). The PartDesc table does not have a corresponding business class defined in the Business Model Workshop.

After the customizations described below, SQL select statements against the Part table will also join with PartDesc. Updates, inserts and deletes will be done on both tables (for example, two queries will be run; one for each table).

➤ **To customize the database mapping of a business class**

1. Create an attribute FullDescrip in class PART in the Business Model Workshop.

    Note that this new attribute will cause a corresponding ATTR_ constant to be generated into class PARTBaseQuery. By default, Express will assume that this new attribute corresponds to a column in table Part; the following customizations will change that.

2. Override the PARTBaseQuery.GetTableName method to cause the BusinessQuery methods to also do selects and updates against the PartDesc table.

    Code Example 2-21   Override PARTQuery.GetTableName

```
method PARTQuery.GetTableName(TableIndex : integer,
      output TableName : TextData)
if (TableIndex = 2) then
  TableName = TextData(value = 'PartDesc');
else
    super.GetTableName(TableIndex = TableIndex,
                       TableName = TableName);
end if;
```

Code Example 2-21   Override PARTQuery.GetTableName *(Continued)*

```
method PARTQuery.GetNumTables() : integer
return (1 + super.GetNumTables());
```

3. Override the PARTBaseQuery.GetColumnName method to handle the attributes for table PartDesc.

This method tells Express that columns Partno and FullDescrip are in the PartDesc table (otherwise, it will think the columns are in the Part table).

Code Example 2-22   Override PARTQuery.GetColumnName

```
method PARTQuery.GetColumnName(attr : integer,
    input output tableIndex : integer,
    output columnName : TextData)
-- For key attribute, set only the columnName.
-- For non-key attributes, set columnName
-- and tableIndex.
if (attr = ATTR_FULLDESCRIP) then
    tableIndex = 2;  -- set table PartDesc
    columnName = 'FullDescrip';
elseif (attr = ATTR_PARTNUMBER) and (tableIndex = 2) then
    -- Required, because key column has different
    -- name in table PartDesc (tableIndex=2).
    columnName = 'PartNo';
else
    super.GetColumnName(attr = attr,
                   tableIndex = tableIndex,
                   columnName = columnName);
    return;
end if;

tableIndex = TableAliases[tableIndex].Value;
```

Using Inheritance in Business Models

This example shows how to customize Express for the case where a set of business classes form an inheritance relationship, and each class is associated with a separate table in the database.

When you specify an inheritance relationship between classes in the Business Model Workshop, Express assumes that all the classes in the inheritance relationship are associated with a single database table. However, you may want to create an application that requires each business class to map to a separate database table.

This example discusses the case where each business class maps to a database table. Each mapped table has the same number of Primary Key columns (the key columns may have different names), and the tables have a one-to-one relationship to each other. Note that every Part is either a CatalogPart or a CustomPart, so the row count in the Part database table is equal to the sum of the row count in both of the Part subclass tables. This configuration is not supported directly by Express and the Business Model Workshop, but can be added as a customization.

The table below shows the descriptions of the tables used in this example:

Table Name	Columns
Part	PartNumber (primary key) Description Type Price
CatalogPart	PartNumber (primary key) CatalogNumber
CustomPart	PartNumber (primary key) AssemblyInstructions

The customization builds on the previous example where we map certain attributes in a business class to another table. This example adds a join table for each of the two subclasses, and maps the subclass attributes to columns in those tables.

This application will display a list of Part objects in an array field. If you double-click on a Part row, Express drills down to another window—the window that opens depends on whether the Part clicked was a CatalogPart or a CustomPart (Express determines this by checking the value of attribute Part.Type, which is not visible in the array field). The business model in the figure below illustrates the relationship between the classes used in this application:

Figure 2-6   Business Model Generalization
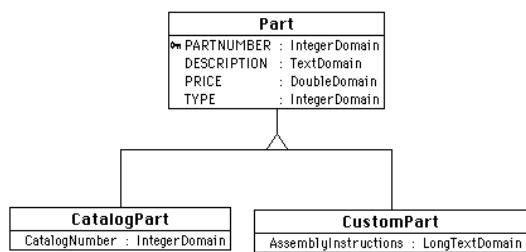
➤ **To create an inheritance relationship between business classes based on different database tables**

1. Create three windows in the Application Model Workshop, one window for each Business Class and create links between these three windows.

   The top-most window is for the Part class and has its Layout of Fields property set to Array. The other two ("linked") windows are for the CatalogPart and CustomPart classes and have a Layout of Fields property of "Form"; these both display all the fields in Part plus the additional fields in the subclass. The Part window has a command link to the other two windows. However, the command link buttons will be made invisible below, and the link will be invoked using a double-click on the array field row.

2. In the BusinessQuery classes CatalogPartQuery and CustomPartQuery, override the GetTableName, GetNumTables, and GetColumnName methods so Express will perform a join with the subclass table, and will know which column is in that class:

```
method CatalogPartQuery.GetTableName(
  TableIndex : integer,
  output TableName : TextData)
if (TableIndex = 2) then
  TableName = TextData(value = 'CatalogPart');
else
  super.GetTableName(TableIndex = TableIndex,
      TableName = TableName);
end if;

method CatalogPartQuery.GetNumTables() : integer
return (1 + super.GetNumTables());

method CatalogPartQuery.GetColumnName(
    attr : integer, input output tableIndex : integer,
    output columnName : TextData)
-- For key attribute, set only the columnName.
-- For non-key attrs, set columnName & tableIndex.
```

```
if (attr = ATTR_CATALOGNUMBER) then
    tableIndex = 2;  -- set table CatalogPart
    columnName = 'CatalogNumber';
elseif attr = ATTR_PARTNUMBER and tableIndex = 2 then
    columnName = 'PartNumber';
else
    super.GetColumnName(attr = attr,
      tableIndex = tableIndex,
      columnName = columnName);
    return;
end if;
tableIndex = TableAliases[tableIndex].Value;

method CustomPartQuery.GetTableName(
  TableIndex : integer,
  output TableName : TextData)
if (TableIndex = 2) then
  TableName = TextData(value = 'CustomPart');
else
  super.GetTableName(TableIndex = TableIndex,
    TableName = TableName);
end if;

method CustomPartQuery.GetNumTable() : integer
return 1 + super.GetNumTables();

method CustomPartQuery.GetColumnName(
    attr : integer, input output tableIndex : integer,
    output columnName : TextData)
if attr = ATTR_ASSEMBLYINSTRUCTIONS then
    tableIndex = 2;
    columnName = 'AssemblyInstructions';
elseif attr = ATTR_PARTNUMBER and tableIndex = 2 then
    columnName = 'PartNumber';
else
    super.GetColumnName(attr = attr,
      tableIndex = tableIndex,
      columnName = columnName);
    return;
end if;
tableIndex = TableAliases[tableIndex].value;
```

3. Override PartWindow.GetRecordTemplate to tell Express to add the "type" column to the target list of any queries this window performs on the Part table:

Code Example 2-23   Override PartWindow.GetRecord Template

```
method PartWindow.GetRecordTemplate(
      template: BusinessQuery = NIL)
       : BusinessQuery
```

Complex Examples

Code Example 2-23   Override PartWindow.GetRecord Template *(Continued)*

```
np : BusinessQuery = super.GetRecordTemplate(
        template = template);

np.AddAttr( attr = NPartQuery.ATTR_TYPE );
return( np );
```

4. Make the command link buttons invisible in PartWindow.

    Since both buttons are contained in the grid field "SideBCGrid", make that grid
    field invisible (see "Setting Widget State" on page 104).

```
method PartWindow.SetWindowState( state : integer)
super.SetWindowState(state = state);

SetWidgetState(<SideBCGrid>, FS_INVISIBLE);
```

5. Handle the childDoubleClick event on the array field in PartWindow. When
    the event is posted, check the type of the part in that row and then invoke the
    appropriate window by invoking its generated CommandLinkTo method.
    Note that the buttons that normally invoke the CommandLinkTo methods
    were made invisible above.

```
eventHandler PartWindow.CustomEvents()
when <DisplayedResultSet>.ChildDoubleClick do
    pc : PartClass = GetCurrentRecord();
    if pc.TYPE.Integervalue = 1 then
        CommandLinkToCatalogPartLink1();
    elseif pc.TYPE.integervalue = 2 then
        CommandLinkToCustomPartLink2();
    end if;
```

Now when you start PartWindow and select data into its array field, you can
double-click on a row and another window will display the subclass information
for the Part in that array field row.

Chapter 2   Customizing Express Applications   145

Restricting the Query to Select a Single Row

You can restrict the queries run by the linked subclass windows so they only select the single row associated with the double-clicked row in the array field. In the above example, the subclass windows select *all* the CatalogPart rows or *all* the CustomPart rows.

To restrict the query, pass search criteria to limit the select to the specific row to the called window. The called window will in turn modify the query to add the criteria to the query's where clause. To pass the restriction to the called window, use LinkInfo.AppData. The following shows how to perform this customization for the PartWindow-to-CatalogPartWindow link (the PartWindow-to-CustomPartWindow link would be customized in the same way).

➤ **To restrict a query to select a single row**

1. Override the CommandLinkToCatalogPartLink1 method to cause the PARTNUMBER in the current record to be passed to the linked window (it is passed as LinkInfo.AppData):

```
method PartWindow.CommandLinkToCatalogPartLink1(
    data : Array of BusinessClass=NIL,
    query : BusinessQuery = NIL,
    appData : Object=NIL) : Array of BusinessClass
return super.CommandLinkToCatalogPartLink1(
    data = data, query = query,
    appData = GetCurrentRecord().PARTNUMBER);
```

2. Override the PreOpenInit method in the linked CatalogPart window and use the PARTNUMBER passed by the calling window as an additional restriction in the query it will subsequently run:

```
method CatalogPartWindow.PreOpenInit()
-- Add additional restriction to query to be run
-- by this window (query is an argument to window).
WindowInfo.InitialSearch.AddConstraint(
  attr = CatalogPartQuery.ATTR_PARTNUMBER,
  value = DataValue(WindowInfo.AppData),
  type = IntegerDomain());
super.PreOpenInit();
```

Providing Automatic Append on Insert in an Array Window

This example is not available from the Customization Manager online Help.

By default, array fields that Express generates do not support the standard automatic append feature of iPlanet UDS, which allows the end user to insert a record into an array field by either clicking in the empty row following the last row non-empty row displayed in the array field or by tabbing from the last column in the last row displayed in the array field. However, you can customize your generated window to simulate this feature.

➤ **To add automatic append functionality**

1. You must add a CustomEvents event handler to handle the event iPlanet UDS posts when an end user clicks on the empty row or tab from the last column. In this event handler, you will add code that creates the new record and includes it in the window's result set.

Code Example 2-24   Create CustomEvents event handler

```
Event Handler LineItemWindow.CustomEvents() : EventRegistration

-- This code allows the user to auto append in an array field.
-- Here we add code to instantiate a new record and include
-- it in the window's result set.
when <DisplayedResultSet>.AfterRowAppend(obj = NewObject) do
  NewObject( BusinessClass(obj) );
  <DisplayedResultSet>.UpdateFieldFromData();
  ResultSet.AppendRow(BusinessClass(obj));
  MaxIndex.Value = MaxIndex.Value + 1;
  SelectRecord(MaxIndex);
```

Example App: CustomClientTutorialApp
• **Project:** CustomClientTutorialAppWindows
• **Class:** LineItemWindow • **EventHandler:** CustomEvents

2. Turn on automatic append in the array field using the AllowsAppend attribute.

    In most cases, you will not want auto append to be on at all times, so you must add code to turn it on and off as appropriate given the current window mode and command set. The way you do this will depend on how you intend to use the window that contains the array, as described below.

If your window runs as a top-level window (that is, it is not nested inside of another window), override the window's SetWindowState method and include the following code:

Code Example 2-25   Override ExpressClassWindow. SetWindowState

```
method MyWindow.SetWindowState(state : integer)

super.SetWindowState(state);

case state is
  when ST_EDIT do
    <DisplayedResultSet>.AllowsAppend = TRUE;

  when ST_SEARCH do
  <DisplayedResultSet>.AllowsAppend = FALSE;

  when ST_VIEW do
    if WindowInfo.CommandSet = CommandMgr.CS_EDIT_MULTIPLE
      or WindowInfo.CommandSet = CommandMgr.CS_ALL then
      <DisplayedResultSet>.AllowsAppend = TRUE;
    else
      <DisplayedResultSet>.AllowsAppend = FALSE;
    end if;
end case;
```

If you use the array window as a nested window that displays multiple aggregate associated records, then you do not need to override SetWindowState. Instead, you will add code to the array window's parent window to decide when the nested window should allow auto append. Override the DisplayCurrentRecord method in the parent window and add the following code:

Code Example 2-26   Override ExpressClassWindow. DisplayCurrentRecord

```
method CustomerOrderWindow.DisplayCurrentRecord()

super.DisplayCurrentRecord();

if WindowMode.Value = WM_EDIT and MaxIndex.Value > 0 then
  LINEITEMLink2Nested.<DisplayedResultSet>.AllowsAppend =
TRUE;
else
  LINEITEMLink2Nested.<DisplayedResultSet>.AllowsAppend =
FALSE;
```

Code Example 2-26   Override ExpressClassWindow. DisplayCurrentRecord

```
end if;
```

Example App: CustomClientTutorialApp
- **Project:** CustomClientTutorialAppWindows
- **Class:** CustomerOrderWindow • **Method:** DisplayCurrentRecord

Project: CustomClientTutorialAppWindows
- **Class:** LineItemWindow • **EventHandler:** CustomEvents

Using Domains

Domains provide a general mechanism for validation code and specialized display of attributes. Do not place field validation code in the SetValue method for the custom domain, because the SetValue method is invoked on the domain class for each attribute when rows are loaded into the DisplayedResultSet array. Thus, SetValue will probably run more often than you want for validation (generally, you only want to validate data entered by the user when she attempts to leave the field).

The fields on the generated window will have mapped attributes that are of the domain type specified for the corresponding business class attribute. The types of the business class attributes will also be domain types. For example, the objects in the DisplayedResultSet array will have attributes whose type is a domain. Thus, because domains can be associated with fields and with business classes, keep the following in mind: the domain (mapped attribute) does not have the widget it is associated with as a class variable. In many cases, there is no associated widget, as with window.DisplayedResultSet (array of BusinessClass), where each attribute is a domain type and there is no directly associated widget.

Even though you cannot assume a domain has a field associated with it, the domain is where you want to place your field manipulation code (for example, list field loading, validation, and so on). So you will sometimes want to make the widget associated with a domain a parameter to domain methods. However, note that if you do this, then you will have to make the Display project a supplier to the domain project.

For example, in the window.PreOpenInit method you might invoke a method on the domain associated with a list field. The method would fill the list field's ListElement array, and would be passed a ListField parameter. For example:

```
stateDomain.SelectList(field = stateList)
```

Selecting into a List Field From a Database Table

This example shows how a domain with a form widget of DropList, RadioList, or ScrollList can be loaded by way of a database query that is run by the Express Services partition.

In this example, we want to create a droplist with part numbers and part descriptions. We have many domains with a similar need, and therefore will create an abstract superclass domain class (SelectListIntegerDomain) as a subclass of IntegerNullable and place the method that performs the query there. We then create our domain class (PartListDomain) as a subclass of SelectListIntegerDomain and give it a form widget of DropList. A window method invokes the domain method that performs the query and passes it the list field to load.

We will place the custom domain classes in a separate project, which should have ExpressServices as a supplier plan. In the business model, we will map the attribute LINEITEM.PARTNUMBER to the custom domain PartListDomain.

1. Create an abstract domain and a method called SelectList.

SelectList will construct and execute the query, and then load the list field. See "Select Queries" for more information on constructing and running Express queries.

NOTE	The domain method cannot create the PARTQuery object because that would make it dependent on the generated Services project, and would create a circular set of supplier plan inclusions (since the generated Services project already includes the Domain project).

```
method SelectListIntegerDomain.SelectList (
    field : ListField, query : BusinessQuery,
    intattr : integer, textattr : integer,
    client : BusinessClient )
-- Execute a database query and use the results to load a ListField
--
-- Parameters:
-- field - the ListField to load
-- query - the Select query to run. This should be a new query;
--      attributes intattr and textattr will be added to this.
-- intattr - ATTR_ number of the integer attribute in each row of
--      the result set returned by "query."
-- textattr - ATTR_ number of the text attribute in each row of
--      the result set returned by "query."
-- businessclient - the caller's BusinessClient. This object's
--      select() method will be invoked to run the query.

mode : integer;
if client.TransActive() then
      mode = ConcurrencyMgr.TR_CONTINUE;
else
      mode = ConcurrencyMgr.TR_SINGLETON;
end if;
returnSet : Array of BusinessClass = client.Select(
      query = query, transactionMode = mode);

list : Array of ListElement = new;
integerValue : IntegerData;

for b in returnSet do
      integerValue = IntegerData(b.GetAttr(intattr));
      list.AppendRow(ListElement(
            integerValue = IntegerValue.Value,
            textValue = b.GetAttr(textattr).TextValue));
end for;

field.SetElementList(list = list);
```

2. Create a window method "LoadPartList" that will invoke the above domain method, passing it the other arguments it needs to run the query.

We create a temporary domain object to invoke the SelectList method on; we cannot use a domain instance in the array field's mapped array because the array field will be empty at this point.

```
method myWindow.LoadPartList ()
-- Load values into DropList array field column.
p : PartListDomain = new;
q : PARTQuery = new;
q.AddAttr(PARTQuery.ATTR_PARTNUMBER);

p.SelectList(field = <DisplayedResultSet[*].PARTNUMBER>,
  query = q,
      intattr = PARTQuery.ATTR_PARTNUMBER,
      textattr = PARTQuery.ATTR_DESCRIPTION,
      client = self.BusinessClient);
```

NOTE The above window method is invoked at window startup in the Window Initialize/After Window is Open customization is the Customization Manager (Application Model):

```
method myWindow.PostOpenInit()
super.PostOpenInit();
LoadPartList();
```

A possible improvement to this example is to do query result caching. For example, cache the results of the above query, and others like it, on the client, or on a shared service. Give each query a unique name to identify it. Then, rather than always running a query as shown above, check the cache first and only run the query if the cache does not already contain the results of that query.

Global Customization

You can add global customizations by subclassing directly from the ExpressServices or ExpressWindows project. Global customizations affect all classes generated from that point forward. In other words, you make global customization when you wish to affect application-wide features, such as window style.

You can add global customizations by creating a project that contains subclasses of specific classes in ExpressWindows and/or ExpressServices (the specific classes are listed below). Using options in the Business Model Workshop and the Application Model Workshop, you specify that Express use these projects to provide the superclasses for the generated classes.

The following sections describe how to perform global customizations on both ExpressWindows and ExpressServices classes. Remember that you may subclass only the classes that are specified below.

Modifying Window Subclasses of ExpressWindows Classes

You should be familiar with the behavior of iPlanet UDS window inheritance before trying to perform global customizations on Express windows. See *A Guide to the iPlanet UDS Workshops* for more information.

You customize subclasses in the ExpressWindows project when you want to modify the style of a window.

➤ **To modify window style**

1. Create a new project (CustomProject).

2. Choose the Supplier Plans command from the File menu and add ExpressWindows as a supplier to the project.

3. Create subclasses of the following classes in CustomProject:

   ❍ ExpressArrayWindow

   ❍ ExpressFormWindow

   ❍ ExpressOutlineWindow

Each subclass must be given the same prefix, such as *New*ArrayWindow, *New*FormWindow, and *New*OutlineWindow. Each subclass must use the same suffix as its superclass. Note that you must create subclasses of all three of the above classes, even if you do not plan to customize all of them.

When you create these new classes, because they are Window classes, the system will automatically generate a Display method for each. Delete this generated Display method in all three classes.

4. Make your customizations to the window in any or all of these window classes.

Note that the three classes correspond to the three styles specified by the Layout of Fields property in the Business Class Window Properties (Form, Array, and Outline). So to customize all generated Form windows, modify *New*FormWindow, and so on.

Be sure to name every widget you add to the window, or the window inheritance mechanism will not work correctly.

5. In the Application Model Workshop, choose the Custom Generation Options command from the File menu.

6. Enter "New" in the Superclass Prefix for Global Customization property.

This property will initially be set to "Express".

7. Choose the Supplier Plans command from the File menu and add CustomWindows as a supplier plan to the application model.

NOTE   Supplier plans added to the Application or Business Model will be added to the supplier plans for the generated project. These supplier plans will not be automatically removed from the generated project if you remove the supplier plan as a supplier to the application model. Thus, if you change your model supplier plans to, for example, pick up customizations from a different project, then you must manually change the supplier plans in the generated project.

You are now finished making your global customization and can regenerate your application model. The generated classes will contain the modifications you made to the corresponding CustomProject.

To use your customizations in future application models, just perform steps 5-7, as described above.

To see an example of this technique, see the Express example program CustomClientTutorialApp. Look at the CustomClientTutorialAppWindows project and its supplier project, CustomWindows.

Customizing Subclasses of ExpressServices Classes

➤ **To customize ExpressServices classes**

1. Create a new project (CustomProject).

2. Choose the Supplier Plans command from the File menu and add ExpressServices as a supplier to the project.

3. Create subclasses of the following superclasses in CustomProject:

   ❍ BusinessClass

   ❍ BusinessClient

   ❍ BusinessDBMgr

   ❍ BusinessQuery

   ❍ BusinessServiceMgr

   Each subclass must be given the same prefix, such as *New*Class, *New*Client, and *New*DBMgr, *New*Query, and *New*ServiceMgr. Each subclass must use the same suffix as its superclass. Note that you must create subclasses of all five of the above classes, even if you do not plan to customize all of them.

4. Make your customizations to any or all of these classes.

5. In the Business Model Workshop, choose the Custom Generation Options command from the File menu.

6. Enter "New" in the Superclass Prefix for Global Customization property.

   This property will initially be set to "Business".

7. Choose the Supplier Plans command from the File menu and add CustomWindows as a supplier plan to the business model.

NOTE	Supplier plans added to the Business Model will be added to the supplier plans for the generated project. These supplier plans will not be automatically removed from the generated project if you remove the supplier plan as a supplier to the business model. For example, if you change your model supplier plans to pick up customizations from a different project, then you must manually change the supplier plans in the generated project.

You are now finished making your global customization and can regenerate your business model. The generated classes will contain the modifications you made to the corresponding CustomProject.

To use your customizations in future business models, just perform steps 5-7, as described above.

Express Example Applications

iPlanet UDS provides a number of example applications that illustrate how to use Express. This appendix provides instructions on how to install the examples, a brief overview of the applications to help you locate relevant examples, and a section describing each example in detail. Typically, you run an example application, then examine it in the various iPlanet UDS workshops to see how it is implemented. You can modify the examples if you wish.

How to Install Express Example Applications

You can run the Express example applications only after you have imported them into your repository, run a database script, modified the service properties in the business models, and regenerated the applications.

Importing the Examples into your Repository

The examples are located in subdirectories under the `FORTE_ROOT/userapp/express/cl#/examples` directory. The example applications are stored as .pex files. You can import them with the Fscript utility or from the Project Workshop. The simplest way to load all the examples is to import them by including the `ins_apps.fsc` script in Fscript. The `ins_apps.fsc` script is located in the `FORTE_ROOT/userapp/express/cl#/examples/install` directory.

➤ **To import the examples**

1. Bring up Fscript in standalone mode and issue the following command:

```
fscript> UsePortable
# In the lines below, replace the '#' in 'cl#' with the
# Compatibility Level for your release of Express.
fscript> setenv FORTE_EP_EX_CL cl#
fscript> SetPath
%{FORTE_ROOT}/userapp/express/cl#/examples/install
fscript> Include ins_apps.fsc
```

This will import all of the Express example applications, compile them, and quit Fscript.

Creating Database Schema and Inserting Data

All of the Express examples rely on database tables. The following are scripts to create these tables and are located in the
`FORTE_ROOT/userapp/express/cl#/examples/install` directory:

- maketut.syb, which uses Sybase's implementation of SQL

- maketut.ora, which uses Oracle's implementation of SQL

- maketut.inf, which uses Informix's implementation of SQL

`Maketut.ora` does not need to be edited. Simply use file redirection with sqlplus. `Maketut.inf` does not need to be edited, but you must specify the test database you wish to use on the dbaccess command line. If you will be using maketut.syb, edit the first line to use an existing database. For example, create a database called testapps, then edit maketut.syb to start with:

```
use testapps
```

Use the standard mechanism for redirecting the maketut file to load the data into your database. The scripts create database schema and insert data related to an order entry system. They are the same tables used by the Tutorial example in *A Guide to Express*.

Modifying Service Properties in the Business Model

Each Express example consists of at least four plans: the business model, *business_model*Services, the application model, and *application_model*Windows. Before you can run an example, you must modify the service properties in the business model so that they are correct for your database. For each example you plan to run, open the Business Model Workshop and double-click on the business service in the Business Class Service List. The Service Properties dialog will open, as shown in Figure A-1.

Figure A-1   Service Properties Dialog
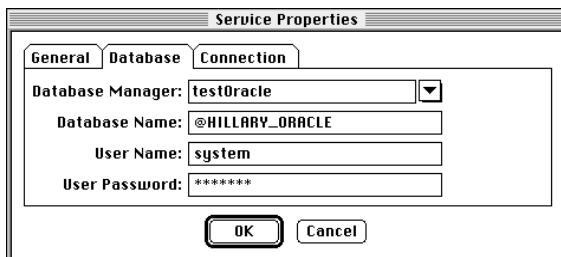
You will need to update the database fields (Database Manager, Database Name, User Name, User Password) with values appropriate for your environment. See *Accessing Databases* if you are unfamiliar with how to reference the Database Manager or Database Name. Once you have updated the fields, click OK.

Regenerating Services

Once you have imported the examples and made the appropriate modification, you need to regenerate the server code. In the Business Model Workshop, select the Generate Server Code command from the File menu. Regenerating the code propagates the new Service Properties you have set into the DBSession service object throughout the *business_model*Services project.

Detailed descriptions of how to run each example are provided in the "Application Descriptions" on page 162.

Removing the Examples

If you want to remove all the examples from your workspace, you can do so by including the `rem_apps.fsc` script in Fscript.

➤ **To remove the examples**

1. Bring up Fscript and issue the following commands:

```
fscript> UsePortable
fscript> SetPath %{FORTE_ROOT}/userapp/express/cl#/examples/install
fscript> Include rem_apps.fsc
```

This will exclude all the Express example applications and quit Fscript.

Removing the Database Tables

When you no longer need the database tables, you can drop them using scripts located in the `FORTE_ROOT/userapp/express/cl#/examples/install` directory. The scripts are the following:

- droptut.syb, which uses Sybase's implementation of SQL

- droptut.ora, which uses Oracle's implementation of SQL

- droptut.inf, which uses Informix's implementation of SQL

`Droptut.ora` does not need to be edited. Simply use standard file redirection with sqlplus. `Droptut.inf` does not need to be edited, but you must supply the test database name on the dbaccess command line. If you will be using droptut.syb, edit the first line to use the database you chose above. For example, if you created a database called testapps, then edit droptut.syb to start with:

```
use testapps
```

Use the standard mechanism for redirecting the maketut file to remove the data from your database.

Overview of Express Example Applications

This section provides an overview of the Express example applications, organized by general topic. The following tables list the example applications under the particular part of the Express system they demonstrate.

The Directory/Example column in each table shows the name of the subdirectory under `FORTE_ROOT/userapp/express/cl#/examples` where you can find the .pex files for the examples, and the name of the example. For the complete description of an individual application, see "Application Descriptions" on page 162.

General-Purpose Express Examples

Directory/Example	Description
tutorial/Tutorial	This example is an on-line copy of the Tutorial described in *A Guide to Express*.

Customized Express Examples—Client

Directory/Example	Description
c_custom/CustomClientTutorialApp	Illustrates how to customize classes generated by the Application Model. Demonstrates many of the client customizations discussed in Chapter 2, "Customizing Express Applications." Uses global customizations and a non-Express login window.
c_custom/CustomClient2App	Illustrates how to make a read-only nested window updateable. Uses an Express window as a login window. And shows how to manipulate data before printing.
c_custom/CustomClient3App	Illustrates how to synchronize data in a modeless linked window.
c_custom/CustomClient4App	Illustrates how to manipulate the result set.
c_custom/CustomClient5App	Illustrates how to access a nested result set. It also shows how to generate records with unique sequence ids.

Directory/Example	Description
c_custom/CustomClient6App	Illustrates how to manipulate the result set in an outline window. It also shows how to delete and display rows in the result set in any window.

Customized Express Examples—Server

Directory/Example	Description
s_custom/CustomQueryApp	Illustrates how to query for table columns not displayed on the window, and how to assign a value to the attribute associated with that table column and save it. It also illustrates how to check if a query is for a certain class, and if the target list contains a certain attribute.
s_custom/CustomQuery2App	Illustrates how to construct your own query. This example performs select and update queries.
s_custom/CustomQuery3App	Illustrates how to construct your own query involving a complex join.

Application Descriptions

This section lists the example applications. Each example has four sections describing it.

The Description section defines the purpose of the example, what problem it solves, and what Express features it illustrates. In all the examples with customizations, a table is provided in the Description section. The table describes:

- the customization features

- the projects, classes, and methods where you can find the code

- where you can find the topic discussed, either in Chapter 2, "Customizing Express Applications," or online, in Customization Manager Help

The online Help location is identified by the notation shorthand given in the table below. You can also access all examples by title from the List of Examples, which is a Help topic in the Customization Manager accessible from both the Business Model and Application Model Workshops. Titles are listed in "Customization Manager Help Files" on page 92.

Some long examples are only available from the List of Examples.

The following table gives the notation used in the Description section tables with the Topic and Item you will find in Customization Manager online Help. (This table does not list all Help Topics or Items, only those referenced in this chapter.)

Notation	Help Topic	Help Item
W/BefLookup	Window	Before Lookup Link
W/Close		Close
W/EventHdlr		Event Handler
W/TabSquence		Modify Tab Sequence
W/SetDisplay'd		Set Displayed Search Criteria
W/SetWState		Set Generated Window State
WI/AfterOpen	Window Initialize	After Window Open
W/InitNested		Initializing as Nested
WD/FieldValCh	Window Data	Field Value Changed
WD/DisplayCR		Display Current Record
WD/NotDisplay		Select Not-Displayed Table Column
WD/ValFields		Validate Fields
WD/ValRecord		Validate Record
T&B/Search	Toolbar and Buttons	Search
T&B/Save		Save
Ap/Start	Application	Start
LOE	List of Examples	
Q/InsUpDel	Queries	Insert/Update/Delete

The Pex Files section gives you the subdirectory and file names of the exported projects. The examples are in subdirectories under the FORTE_ROOT/userapp/express/cl#/examples directory. In general, you should follow the instructions in "How to Install Express Example Applications" on page 157 to install all the examples. You can also import example applications individually if you wish. When multiple .pex files are listed, there are supplier projects in addition to the main project. You will need to import all the files listed to run the application. Be sure to import the files in the order given so that dependencies will be satisfied.

In each example subdirectory you will find at least four files with the following name scheme:

- name_b.pex —the .pex file for the Business Model

- name_a.pex—the .pex file for the Application Model

- name_s.pex—the .pex file for the Services project generated by the Business Model

- name_w.pex—the .pex file for the Windows project generated by the Application Model

Additional files are supplier projects such as custom domains and custom windows or message catalogs.

The Special Requirements section identifies any special setup procedures you may need to follow.

Finally, the To Use section tells you how to step through the application's functions.

CAUTION   Every example requires standard setup steps. Even if you only want to import one of the example applications, you will still need to follow the instructions given in "How to Install Express Example Applications" on page 157 for building database schema, updating service properties, and regenerating the application.

Tutorial

Description   Tutorial is an on-line copy of the application described in the Tutorial chapter in *A Guide to Express*. We recommend you go through the Tutorial chapter to create your own application. If you need to, you can load the Tutorial example we provide and run and examine it. Many of the other Express examples use the Tutorial example as their starting point.

Pex Files   tutorial/tut_b.pex, tutorial/tut_a.pex, tutorial/tut_s.pex, tutorial/tut_w.pex.

Special Requirements   None.

➤ **To use the Tutorial application**

1.   Open the TutorialApp Application Model and click the run icon.

Procedures for querying, updating, inserting, and deleting records are described in the Tutorial chapter in *A Guide to Express*.

CustomClientTutorialApp

Description   CustomClientTutorialApp illustrates numerous customization techniques for the client side of your application. The application:

• has a login window that is called before the application comes up

• uses window inheritance and Express global customization techniques to display a standard logo on all the windows

• checks whether windows are nested, and doesn't display the logo if they are

• illustrates record validation

• illustrates field validation using a custom domain

The table below outlines all the customizations found in this example.

Customization Feature	Project	Class	Method/EventHandler	See...
Adding Processing Before/After a Search	CustomClientTutorial AppWindows	CustomerOrder Window	Search	T&B/Search
Adding Processing Before/After a Save	CustomClientTutorial AppWindows	CustomerOrder Window	Save	T&B/Save
Placing Default Search Criteria in a Field	CustomClientTutorial AppWindows	CustomerOrder Window	ClearFieldsForSearch	W/SetDisplay'd
Detecting Select of New Record	CustomClientTutorial AppWindows	CustomerOrder Window	DisplayCurrent Record	WD/DisplayCR
Changing Generated Component States	CustomClientTutorial AppWindows	CustomerOrder Window	SetWindowState ClearFieldsForSearch	W/SetWState
Setting a Window to Search Mode	CustomClientTutorial AppWindows	CustomerOrder Window	PostOpenInit	WI/AfterOpen
Determining If a User has Changed Data	CustomClientTutorial AppWindows	CustomerOrder Window	Save	page 106
Handling Events	CustomClientTutorial AppWindows	CustomerOrder Window	CustomEvents	W/EventHdlr
Adding Processing If a Window is Nested	CustomWindows	LogoArray Window	InitializingAs Nested()	W/InitNested
Closing the Current Window	CustomClientTutorial AppWindows	CustomerOrder Window	CustomEvents	W/Close

Customization Feature	Project	Class	Method/EventHandler	See...
Using a DrillDown Link to a Callout to Close an Outline Window	CustomClientTutorial AppWindows	PartBaseWindow	DrillDownLink	page 108
Removing a Field from the Tab Sequence	CustomClientTutorial AppWindows	CustomerOrder Window	AddFieldsToTab Sequence	W/TabSquence
Providing Auto Append on Insert in a Nested Array Window	CustomClientTutorial AppWindows	CustomerOrder Window	DisplayCurrent Record	page 147
		LineItemWindow	CustomEvents	
Using a Domain to Validate a Field	CustomClientTutorial AppWindows	LineItemWindow	ValidateField	WD/ValFields
	CustomTutDomains	PosNumDomain	Validate	
Validating a Record	CustomClientTutorial AppWindows	CustomerOrder Window	ValidateRecord	WD/ValRecord
Modifying a SQL Query to Validate a Field	CustomClientTutorial AppWindows	LineItemWindow	ValidateField	WD/ValFields
Calculating a Derived Field	CustomClientTutorial AppWindows	LineItemWindow	FieldValueChanged	WD/FieldValCh
Calculating a Derived Field from Nested Window Data	CustomClientTutorial AppWindows	CustomerOrder Window	CountQuantity	page 133
			DisplayCurrent Record	
			CustomEvents	
		LineItemWindow	QuantityChanged	
			FieldValueChanged	
			DeleteRecordFrom ResultSet	
Using a non-Express Window as a Login Window	CustomClientTutorial AppWindows	CustomClient TutorialAppBroker	RegisterWindow	Ap/Start
	CustomWindows	CustomLogin Window	Display	

Customization Feature	Project	Class	Method/EventHandler	See...
Passing Arguments to Linked Windows	CustomClientTutorial AppWindows	CustomerOrder Window	BeforeLookupLink	W/BefLookup
			PostOpenInit	
		CustomerWindow		
Global Customization of Windows	CustomClientTutorial AppWindows	CustomerOrder Window		page 153
		LineItemWindow		
	CustomWindows	LogoArray Window		
		LogoFormWindow		
		LogoOutline Window		

Pex Files   c_custom/ctutdom.pex, c_custom/custwin.pex, c_custom/cctut_b.pex, c_custom/cctut_a.pex, c_custom/cctut_s.pex, c_custom/cctut_w.pex.

Special Requirements   None.

➤ **To use CustomClientTutorialApp**

1. Open the CustomClientTutorialApp Application Model and click the run icon.

    You will see a Login window.

2. To continue to the application, enter 'Rosebud' and click the OK button.

    To see the Login window fail, enter a bad password twice.

3. Notice that the CustomerOrder window came up in Search mode, rather than Edit mode, which is the default.

4. As you experiment with the application, watch the console window. At a number of customization points, a message will be written to the logger.

5. Notice the logo displayed at the top of the CustomerOrder window and the Customer window. Notice that there is no logo above the nested LineItem window.

6. In the Customer window, notice that it has a different title, which has been passed to it programmatically.

7. In the CustomerOrder window, notice that the Address field is not displayed. It has been turned off programmatically.

8. Go into Insert Mode and insert a record. Try entering a negative value in the quantity field in the LineItem array. You will see an error message when you tab out of the field.

9. Enter several LineItem rows with quantities that, when totalled, add up to more than 100. You will see an error message when you try to save this record.

10. Notice that when you go into Search mode, a Search criteria is placed in the Customer Number field.

11. When you search for records or insert them, two derived fields are updated. The Cost field is based on the Price of the Part and the Quantity in the LineItem array. The ItemCount field is calculated based on the total of the values of the Quantity fields. These fields are kept current whenever records are inserted, deleted, or changed.

12. Enter new LineItem rows. Enter an invalid value for the Part Number. You will see an error message. Enter a valid value ('90003'). This should be allowed.

13. Enter another new LineItem row. This time, leave the Part Number blank, but click on the Lookup button to the right of the Part Number field. Select a Part record in the outline field and double click on it. You should be returned to the LineItem array, and the Part Number and Price fields should be filled in.

14. When tabbing around the CustomerOrder window, notice that the Requested Date field is skipped. That field was removed from the tab sequence programmatically since a default value is provided for the date.

15. When inserting LineItems in the array, notice that Auto Append is turned on. This means you can insert new rows simply by tabbing out of the last field on the prior row, or by clicking in the new row. It is not necessary to click the Insert button.

16. Finally, exit the application by clicking the Exit Now button.

CustomClient2App

Description   CustomClient2App illustrates the use of an Express window as a login window, and shows how to make a read-only nested window updateable. It also shows how to print an Express window. It illustrates the case where you simply want to print the window, and also illustrates how to manipulate data before printing.

Customization Feature	Project/Class	Methods/EventHandlers	See...
Using an Express Window as a Login Window	CustomClient2AppWindows	CommandLinkToCUSTOMERLink1	LOE, page 138
	OrdUserWindow class	AfterCustomerLink1Open	
		StartMethod	
Making a Read-Only Nested Window Updateable	CustomClientTutorial	(examine links in Business and Application models)	
	CustomClient2App		
Printing an Express Window	CustomClient2AppWindows	CustomEvents	
	CustomerWindow	PrintWindow	
Manipulating Data Before Printing an Express Window	CustomClient2AppWindows	CustomEvents	
	CustomerWindow	PrintWindowWithChanges	

Pex Files   c_custom/ctutdom.pex, c_custom/cctut_b.pex, c_custom/cc2_a.pex, c_custom/cctut_s.pex, c_custom/cc2_w.pex.

Special Requirements   None.

➤ **To use CustomClient2App**

1. Open the CustomClient2App Application Model and click the run icon. You will see a Login window. To continue to the application, enter 'Rita', 'Nick', or 'Warren' as the User Name, and 'Rosebud' as the password. Case is significant. Click the Log On button. To see it fail, enter a bad password.

2. Notice that the Customer Order nested array window is updateable, in spite of the fact that it does not have an aggregate relationship with its parent, the Customer window.

3. In the Customer window, select all customers. The first record should have an address and phone number. Click the Print Window button. The printed output should show the data exactly as it appears on the screen.

4. Then click the Print (No Address/Phone) button. Notice in the printed output that the data in both the Address and Phone fields has been replaced with the text 'Withheld for Privacy'.

CustomClient3App

Description   CustomClient3App illustrates how to synchronize data in a modeless linked window.

Customization Feature	Project/Class	Methods/EventHandlers	See...
Synchronizing Data in a Modeless Linked Window	CustomClient3AppWindows	AfterCUSTOMERORDERLink1Open	LOE, page 138
	CustomerWindow class	DisplayCurrentRecord	

Pex Files   c_custom/ctutdom.pex, c_custom/cctut_b.pex, c_custom/cc3_a.pex, c_custom/cctut_s.pex, c_custom/cc3_w.pex.

Special Requirements   None.

➤ **To use CustomClient3App**

1. Open the CustomClient3App Application Model and click the run icon. You will see a Customer window. Search for a customer or customers.

2. Open the CustomerOrder window with the Lookup button.

3. Use the arrow keys in the Customer window to scroll through your result set. Notice the values displayed in the CustomerOrder window change as the Customer changes in the Customer window.

CustomClient4App

Description CustomClient4App illustrates how to manipulate the results set.

Customization Feature	Project/Class	Methods/EventHandlers	See...
Setting the Value of a Displayed Field	CustomClient4AppWindows CustomerWindow class	CustomEvents	page 114
Checking Query Information on a BusinessClass	CustomClient4AppWindows CustomerWindow class	CustomEvents	page 117
Looping Through a Displayed Result Set	CustomClient4AppWindows CustomerWindow class	Custom Events	page 118

Pex Files c_custom/ctutdom.pex, c_custom/cctut_b.pex, c_custom/cc4_a.pex, c_custom/cctut_s.pex, c_custom/cc4_w.pex.

Special Requirements None.

➤ **To use CustomClient4App**

1. Open the CustomClient4App Application Model and click the run icon. You will see a Customer window. Go into Search mode and search for all customers.

2. Click the Check Query button. A message will be written to the logger stating that the business class has not been updated.

3. Click the Change Address button. You will see a new street address displayed in the ADDRESS field.

4. Click the Check Query button again. The logger message will now state that the business class and the address field have been updated.

5. Click the Reset button.

6. Click the Check Query button. A message will be written to the logger stating that the business class has not been updated.

7. Click the Loop Thru button. For each customer record, a message will be written to the logger, stating whether the CustomerNumber is less than or greater than 100.

CustomClient5App

Description   CustomClient5App illustrates how to access a nested result set. It also shows how to generate unique sequence ids for part of a composite key in a component table of a one-to-many aggregate association.

Customization Feature	Project/Class	Methods/EventHandlers	See...
Accessing a Nested Result Set	CustomClient5AppWindows CustomerOrderWindow class	CustomEvents	page 120
Generating Records with Unique Sequence IDs	CustomClient5AppWindows CustomerOrderWindow class	Save Sequence	LOE, page 136

Pex Files   c_custom/ctutdom.pex, c_custom/cctut_b.pex, c_custom/cc5_a.pex, c_custom/cctut_s.pex, c_custom/cc5_w.pex.

Special Requirements   None.

➤ **To use CustomClient5App**

1. Open the CustomClient5App Application Model and click the run icon. You will see a CustomerOrder window with a nested LineItem window. Go into Search mode and search for all customer orders.

2. Scroll through your result set until you find a Customer Order record that has a few LineItem records. Click the Insert button below the LineItem array. Add a couple records. The LineItemNumber column is read-only. Just supply values for the Quantity and PartNumber fields. Add at least one PartNumber 90001.

3. Click the Save icon in the CustomerOrder window. When the record is saved, notice that sequential LineItemNumbers have been assigned. The first new LineItemNumber is always the highest existing LineItemNumber plus one.

4. Experiment with inserting and deleting LineItems.

5. Click the Loop Thru button. This will loop through the LineItem array for the current CustomerOrder record, and write a message to the logger each time it finds a LineItem record with a PartNumber of 90001.

CustomClient6App

Description   CustomClient6App illustrates how to manipulate the results set in an ExpressOutlineWindow and how to remove or display a row in a result set in any window.

Customization Feature	Project/Class	Methods/EventHandlers	See...
Using Displayed Result Sets with Outline Fields	CustomClient6AppWindows CustomerWindow class	CustomEvents	page 118
Removing Rows from a Result Set	CustomClient6AppWindows CustomerWindow class	CustomEvents	page 119
Displaying a Row in a Result Set	CustomClient6AppWindows CustomerWindow class	Custom Events	page 119

Pex Files   c_custom/ctutdom.pex, c_custom/cctut_b.pex, c_custom/cc6_a.pex, c_custom/cctut_s.pex, c_custom/cc6_w.pex.

Special Requirements   None.

➤ **To use CustomClient6App**

1. Open the CustomClient6App Application Model and click the run icon. You will see a Customer outline window. Go into Search mode and search for all customers.

2. Click the Loop Thru button. A message will be written to the logger stating which customer numbers are greater than or less than 100.

3. Click the Add Row button. You will see a new row displayed.

4. Select a row. Click the Remove row button. The row will disappear.

5. Click the Display Row 2 button. The second row in the result set will be highlighted.

CustomQueryApp

Description CustomQueryApp illustrates how to query for table columns not displayed on the window, and how to assign a value to the attribute associated with that table column and save it. It also illustrates how to check if a query is for a certain class, and if the target list contains a certain attribute.

Customization Feature	Project/Class	Methods/EventHandlers	See...
Selecting a Table Column not Displayed on a Window	CustomQueryAppWindows CustomerWindow class	GetRecordTemplate	WD/Not Display
Checking if a Query is for a certain class	CustomQueryAppWindows CustomerWindow class	GetRecord Template	page 121
Checking if a Target List Contains certain Attributes	CustomQueryAppWindows CustomerWindow class	GetRecord Template	page 121
Using the LogAttr method to Save Attributes not Displayed on a Window	CustomQueryAppWindows CustomerWindow class	Save	page 115

Pex Files c_custom/ctutdom.pex, c_custom/cctut_b.pex, s_custom/cq_a.pex, c_custom/cctut_s.pex, s_custom/cq_w.pex.

Special Requirements None.

➤ **To use CustomQueryApp**

1. Open the CustomQueryApp Application Model and click the run icon. You will see a Customer form window which is missing the address field.

2. Examine the console window. Information about the query and the target list will be written there.

3. Enter values for a new customer. Save this record, and make a note of the CustomerNumber.

4. Exit CustomQueryApp.

5. You may now want to confirm that a value for address field was assigned programmatically. You can do this by directly querying the Customer table in your database, or by running the Tutorial example, opening the Customer window in that application and searching for the Customer you just added. You should see the value 'River Road' in the address field.

CustomQuery2App

Description CustomQuery2App illustrates how to construct your own query. It executes select and update queries.

Customization Feature	Project/Class	Methods/EventHandlers	See...
Constructing a New Select Query	CustomQuery2AppWindows CustomerWindow class	ComposeQueries	page 124
Constructing a New Update Query	CustomQuery2AppWindows CustomerWindow class	ComposeQueries	page 126

Pex Files c_custom/ctutdom.pex, c_custom/cctut_b.pex, s_custom/cq2_a.pex, c_custom/cctut_s.pex, s_custom/cq2_w.pex.

Special Requirements None.

➤ **To use CustomQueryApp**

1. Open the CustomQuery2App Application Model and click the run icon. You will see a Customer form window.

2. Examine the console window. The constructed queries are run automatically after the window opens. Information about the customer records will be written there. You will see the customer numbers returned by the select statement, the customer numbers after three new customers are added to the CUSTOMERClass array, and the customer numbers after one new customer has been deleted and the update has been executed.

3. In the CUSTOMER window, go into Search mode and select all customers. As you scroll through the records, you should see the two new customers.

CustomQuery3App

Description   CustomQuery3App illustrates how to construct your own query involving a complex join. It executes only a select query.

Customization Feature	Project/Class	Methods/EventHandlers	See...
Constructing a New Select Query involving Complex Join	CustomQuery3AppWindows CustomerOrderWindow class	ComposeQueries	page 124

Pex Files   c_custom/ctutdom.pex, c_custom/cctut_b.pex, s_custom/cq3_a.pex, c_custom/cctut_s.pex, s_custom/cq3_w.pex.

Special Requirements   None.

➤ **To use CustomQueryApp**

1. Open the CustomQuery3App Application Model and click the run icon. You will see a CustomerOrder form window.

2. Examine the console window. The constructed query is run automatically after the window opens. The results of the select query, which joins four tables, will be written there.

Index