TOOL Reference Manual

Release 3.5 of Forte™ 4GL

Sun Microsystems, Inc., 901 San Antonio Road, Palo Alto, CA 94303 U.S.A. 1-800-786-7638

Part No. 806-6674-01
October 2000, Revision A

Contents

Preface

The *TOOL Reference Manual* provides complete information on Forte's transactional, object-oriented language called TOOL.

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

- have programming experience
- are familiar with your particular window system
- are familiar with SQL and your particular database management system
- understand the basic concepts of object-oriented programming as described in *A Guide to the Forte 4GL Workshops*
- have used the Forte Workshops to create classes

Organization of This Manual

This manual is organized primarily for reference. We assume that you are using the Forte workshops to create custom classes, and need information about TOOL in order to write methods, event handlers, cursors, and virtual attributes.

This manual presents a brief overview of the TOOL language, followed by details about the language elements and statements. It contains the following chapters:

Chapter	Description
Chapter 1, "Overview"	Defines the basic concepts of the language and provides an overview of its features.
Chapter 2, "Language Elements"	Provides detailed information about the TOOL language elements, including statements, comments, objects, variables, and so on.
Chapter 3, "TOOL Statement Reference"	Provides detailed information about the TOOL statements for writing methods. This chapter is organized alphabetically by statement.
Chapter 4, "Project Definition Statements"	Provides detailed information about the Forte project definition statements for use with the Fscript utility. This chapter is organized alphabetically by statement.
Appendix A, "Reserved Words"	Contains a list of reserved words for TOOL and SQL.
Appendix B, "Forte TOOL Example Applications"	Provides a number of example applications that illustrate how to use the features described in this manual.]

Note on the TOOL SQL statements

A subset of TOOL statements (the TOOL SQL statements) are described briefly in Chapter 3, "TOOL Statement Reference." The manual *Accessing Databases* is the primary reference source for this set of statements.

Conventions

This manual uses standard Forte documentation conventions in specifying command syntax and in documenting TOOL code.

Command Syntax Conventions

The specifications of command syntax in this manual use a “brackets and braces” format. The following table describes this format:

Format	Description
bold	Bold text is a reserved word; type the word exactly as shown.
italics	Italicized text is a generic term that represents a set of options or values. Substitute an appropriate clause or value where you see italic text.
UPPERCASE	Uppercase text represents a constant. Type uppercase text exactly as shown.
underline	Underlined text represents a default value.
vertical bars |	Vertical bars indicate a mutually exclusive choice between items. See braces and brackets, below.
braces { }	Braces indicate a required clause. When a list of items separated by vertical bars is enclosed in braces, you must enter one of the items from the list. Do not enter the braces or vertical bars.
brackets [ ]	Brackets indicate an optional clause. When a list of items separated by vertical bars is enclosed by brackets, you can either select one item from the list or ignore the entire clause. Do not enter the brackets or vertical bars.
ellipsis …	The item preceding an ellipsis may be repeated one or more times. When a clause in braces is followed by an ellipsis, you can use the clause one or more times. When a clause in brackets is followed by an ellipsis, you can use the clause zero or more times.

TOOL Code Conventions

Where this manual includes documentation or examples of TOOL code, the TOOL code conventions in the following table are used.

Format	Description
parentheses ( )	Parentheses are used in TOOL code to enclose a parameter list. Always include the parentheses with the parameter list.
comma ,	Commas are used in TOOL code to separate items in a parameter list. Always include the commas in the parameter list.
colon :	Colons are used in TOOL code to separate a name from a type, or to indicate a Forte name in a SQL statement. Always include the colon in the type declaration or statement.
semicolon ;	Semicolons are used in TOOL code to end a TOOL statement. Always type a semicolon at the end of a statement.

The Forte Documentation Set

Forte produces a comprehensive documentation set describing the libraries, languages, workshops, and utilities of the Forte Application Environment. The complete Forte Release 3 documentation set consists of the following manuals in addition to comprehensive online Help.

Forte 4GL

- *A Guide to the Forte 4GL Workshops*
- *Accessing Databases*
- *Building International Applications*
- *Escript and System Agent Reference Manual*
- *Forte 4GL Java Interoperability Guide*
- *Forte 4GL Programming Guide*
- *Forte 4GL System Installation Guide*
- *Forte 4GL System Management Guide*
- *Fscript Reference Manual*
- *Getting Started With Forte 4GL*
- *Integrating with External Systems*
- *Programming with System Agents*
- *TOOL Reference Manual*
- *Using Forte 4GL for OS/390*

Forte Express

- *A Guide to Forte Express*
- *Customizing Forte Express Applications*
- *Forte Express Installation Guide*

Forte WebEnterprise and WebEnterprise Designer

- *A Guide to WebEnterprise*
- *Customizing WebEnterprise Designer Applications*
- *Getting Started with WebEnterprise Designer*
- *WebEnterprise Installation Guide*

Forte Example Programs

In this manual, we often include code fragments to illustrate the use of a feature that is being discussed. If a code fragment has been extracted from a Forte example program, the name of the example program is given after the code fragment. If a major topic is illustrated by a Forte example program, reference will be made to the example program in the text.

These Forte example programs come with the Forte product. They 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.

The Forte example programs are installed automatically in the Forte development environment. If they are not there, ask your system administrator to install them. The procedure for installing the examples is documented in the *Forte 4GL System Installation Guide*. You can run the examples in the Project Workshop, experiment with using them, run them under the Debugger, and make changes to the TOOL code.

The example programs that illustrate the features covered by the *TOOL Reference Manual* are described in Appendix B, “Forte TOOL Example Applications.” This appendix describes the purpose of each example, and how to run it.

Viewing and Searching PDF Files

You can view and search 4GL PDF files directly from the dcumentation 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 fortedoc directory and its contents from the CD-ROM to the client or server hard disk.

   You can specify any convenient location for the fortedoc directory; the location is not dependent on the Forte distribution.

2. Set up a directory structure that keeps the fortedoc.pdf and the 4gl 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 fortedoc directory.

▶ **To view and search the documentation:**

1. Open the file fortedoc.pdf, located in the fortedoc 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+]

6. To return to the fortedoc.pdf file, click the Homepage bookmark at the top of the bookmarks list.
7. To revisit the query results, click the **Results** button at the bottom of the fortedoc.pdf home page or select **Edit** > **Search** > **Results**.

Overview

This chapter describes how to use Forte's programming language, TOOL. It provides an overview of the language, and discusses the following basic programming concepts:

- object-oriented programming
- event-based programming
- multitasking
- transactions
- interacting with the database
- exception handling

What is TOOL?

TOOL is an object-oriented programming language, especially designed for programming in a distributed environment. In Forte, you use TOOL to write methods, which provide the basic operations for your application. You also use TOOL to write named event handlers, cursors, and virtual attribute expressions.

Manipulating objects

Using TOOL, you can manipulate any of the objects in your application. You create the objects your application needs by using object constructors. You can access and set the object attributes, and you can invoke methods and post events on the objects. To see a discussion of TOOL language elements, refer to Chapter 2, "Language Elements."

Programming statements

TOOL provides programming statements for performing the processing within a method. These statements, which fall into the following general categories, are documented in Chapter 3, "TOOL Statement Reference."

Category	Description
Declaration and assignment	Declaration and assignment statements declare local variables and assign them values.
Control flow	Standard control statements affect the flow of a program and include the **case**, **for**, **if**, and **while** statements.
Event handling	The **post** statement triggers an event. The **event** statement provides processing in response to one or more specified events. The **register** statement includes named event handlers within the current **event** statement.
Multitasking	The **start task** statement starts a new task.
Transactions	The **begin transaction/end transaction** block executes the specified code as a transaction.
Database access	SQL Data Manipulation Language statements query and update relational database tables. The **sql execute immediate** statement lets you execute any SQL statement and **sql execute procedure** lets you execute a database procedure.
Exception handling	The **exception** clause provides exception handling for the current statement block. The **raise** statement generates an exception.

Variables

Forte variables contain or point to the data that your application manipulates. All information that you display in windows, accept as input from end users, or retrieve from a database or other outside source is stored in variables. Variables have either a simple type (such as integer or string) or a class type. If the variable has a simple data type, the variable itself contains the data. If the variable has a class type, the variable points to the object that contains the data.

You must declare a variable in your TOOL code before you can reference it. After you have declared the variable, you can assign a value to it, reference it, or include it in expressions.

Forte classes

Forte provides many predefined classes in the Forte libraries. The Forte Framework library provides important functionality, and includes classes that provide special data structures, classes that provide runtime information about the state of the system, classes for interacting with database management systems, and classes for using transactions. The Forte Display library classes allow you to manipulate the user interface for the application, including the windows, fields, and menus. See the Framework Library online Help and Display Library online Help manuals for more information.

Project definition statements

Normally you create projects interactively in the Project Workshop. However, you can also use Forte's Fscript utility (described in *Fscript Reference Manual*) to declare classes and other project components in an external file. In Fscript you use TOOL project definition statements, such as **begin**, **class**, **constant**, **cursor**, **event handler**, **interface**, **method**, and **service**, to define various project components. For reference information regarding these statements, see Chapter 4, "Project Definition Statements."

Object-Oriented Programming

In the Project Workshop, you define the custom classes for your application. A vital part of creating these classes is writing the methods associated with them. In an object-oriented system, methods are the “procedures” that run the application. The startup method for a project begins execution of the application by allocating a startup object and then operating on it. This method not only performs operations on the object, but it invokes other methods. Each method that it invokes may, in turn, invoke other methods. This process continues until the startup method completes. When the startup method completes, the application exits.

Object-oriented programming in TOOL means writing new methods for your custom classes. Within these new methods, you operate on objects by changing their attributes and invoking other methods. The methods you invoke can be from prefabricated classes or from other custom classes.

Forte provides many useful classes and methods that you can use in your own methods. Be sure to study the classes in the Forte manuals, particularly Framework Library online Help and Display Library online Help, before you write any custom methods.

Constructing Objects

In Forte, every object is associated with a *data item*. The data item can be a variable, an attribute, or a parameter, and it serves as a reference to the object. You use the data item’s name to reference the object. For example, in Figure 1 the variable “a” refers to an object of class Artist:

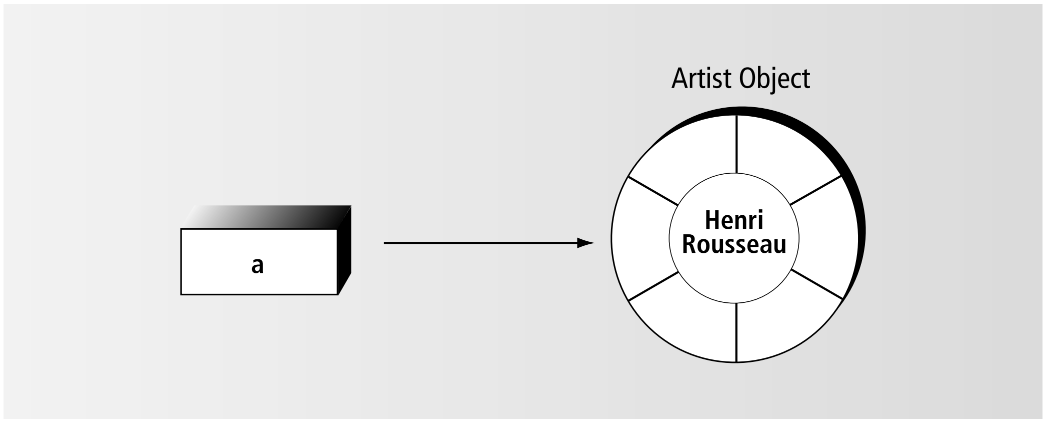


Figure 1 *Class Type Variable*

Windows are objects

Because windows are objects in Forte application, you also use object-oriented programming techniques to structure the user interface for your application. To manipulate the windows in an application, you use the methods for the UserWindow class.

Object constructors

Before you can manipulate an object in your method, you must create it by using an object constructor. An object constructor creates a new object and specifies its initial values. For more information on object constructors, refer to “Using Object Constructors” on page 65.

Variables

For a variable, you can construct the object that the variable points to either while you are declaring the variable or when you are assigning a value to the variable. The following example illustrates constructing the object while declaring the variable:

Example: object constructors for variables

```
t : TextData = new(value = ‘Hello World’);
a : Artist = new(Name = TextData(value =
    ‘Henri Rousseau’),Born = 1844, Died = 1910);
```

Attributes and parameters

For attributes and parameters, you can construct a new object only when you are assigning a value to it.

```
self.Window.Title = TextData(value = 'New Title');
self.AuctionManager.DeleteBidForPainting (name = new(value =
'mona'));
```

See Chapter 2, "Language Elements," for details about constructing objects.

Writing Methods

You use the Class Workshop to create (name) a new method, define its parameters and return value (if any). Then you use the Method Workshop, as described in *A Guide to the Forte 4GL Workshops* to write the method using TOOL statements.

Statements and comments

The method body consists of TOOL statements and comments. Within an individual method, you can provide many different kinds of processing, such as handling events, starting transactions, initiating new tasks, and changing attributes. The TOOL statements that you use to provide this processing are briefly described in this chapter. Full details are available in Chapter 3, "TOOL Statement Reference."

Local variables

You can declare local variables anywhere in the method body. The scope of a local variable is from the point you declare it to the end of the method. The following example illustrates declaring integer and TextData variables:

```
i : integer;
t : TextData;
```

Specifying a return value

If a method has a return type, one element is required. You must use the **return** statement to end the method and specify the return value.

In the following example, the return type for the method is defined as an integer. The **return** statement in the method specifies a value of 10, which Forte passes back to the invoking method. The invoking method can then assign this return value to a variable or use it in any appropriate expression.

```
return 10;
```

If a method does not have a return type, you do not use the **return** statement. When the method completes, control automatically returns to the invoking method.

Invoking Methods

To invoke a method, you specify the object to be operated on and the name of the method. If the method has parameters, you must pass a value for each required parameter (that is, for each parameter that was not defined with a default value). The following example illustrates two ways to invoke a method for the TextData class:

Example: invoking methods

```
t : TextData = new;
-- Using named parameters
t.SetValue(source = ‘Hello’);
-- Using positional parameters
t.SetValue(‘Hello’);
```

Methods are like procedures

Invoking a method is the same as calling a procedure or a subroutine. When one method invokes another method, control passes to the invoked method. When the invoked method completes, control returns to the invoking method. Unless you use multitasking, methods are executed synchronously. There is a single flow of control.

For details about referencing objects and invoking methods, see “Referencing an Object” on page 67 and “Invoking Methods” on page 73.

Manipulating Attributes

When you need to report current object values, update object values, or perform calculations using object values, you work directly with the object’s attributes. Within a method, you can access or set the value of any of the current object’s attributes.

Example:
setting attribute value

```
a : Artist = new;
-- Simple type attribute
a.Born = 1844;
-- Object ref attribute
a.Name = new(value = ‘Vincent Van Gogh’);
```

Working with a virtual attribute is exactly the same. You simply access and set its value without worrying about how it is implemented. For details about accessing and setting attributes, see “Referencing an Object” on page 67 and “Setting Attributes” on page 72.

Event-Based Programming

Event-based programming means providing code that is executed when a particular event occurs and using events to signal other windows and processes when something relevant happens. For example, the Display method for a window can include code that is executed when the end user selects a menu item, clicks a button, or enters data into a text field.

Forte lets you respond to numerous kinds of events, including events in the distributed environment and custom events. Forte shared services use events to communicate with clients and with other services.

At any time, a service can post an event to the rest of the application. This is like “broadcasting” a signal to the rest of the application that a certain condition or action has occurred. Any number of clients or services may be “tuned in,” waiting for that particular signal. If so, when they receive the event, they will respond to it. Events eliminate the timing problems associated with polling, and reduce the use of system resources.

Forte provides three statements for working with events:

Statement	Description
post	Produces an event
event	Provides processing in response to one or more events.
register	Includes a named event handler in an **event** statement.

Events

An event is a notification that a particular action has occurred. This can be an action such as the end user selecting a menu item or a node going offline.

Every event is associated with an object. The class for an object defines the types of events that the object can produce. For example, the TextField class defines an AfterValueChange event, which is triggered when the end user leaves the field after changing the data in it.

Every event has a name, which you use to identify the event when you want to post it or respond to it. Some events have parameters, to provide information about the particular occurrence of the event. For example, the AfterMove event for fields includes two parameters that indicate the X and Y positions of the field before it was moved. You can use this information when you respond to the event.

Display library events

The Display library classes, described in the manual *Display Library online Help*, define the events that can be triggered by end user actions in the user interface. For example, the PushButton class defines a Click event, which is triggered when the end user clicks the push button. These kinds of events include events on windows, such as closing and iconizing, as well as events on individual widgets, such as data entry in a text field and selection of a menu item. Typically, the Display methods for your windows respond to these kinds of events.

Other system events

Other Forte system classes define events that occur throughout the distributed environment. An example is the event RemoteAccessEvent on the Object class, which is posted when a distributed reference loses contact with the remote object that it references. If your application is waiting for a particular event on a remote service object in order to continue processing, the RemoteAccess event would notify you when contact with the remote service object is lost. In response to this event, your application could recover and allow the end user to take alternate action.

All Forte system events are triggered automatically when the appropriate action occurs. You can also trigger them using the **post** statement within a method. See Framework Library online Help for information on the individual events for the system classes.

Custom events

The custom classes that you create for an application (or include with supplier plans) may also have events. Custom events are not triggered automatically. At the appropriate point in a method, you must trigger the custom event by using the **post** statement. For example, in the Auction sample application, the BidUpdated event is defined for the Bid class. Every time a bid is updated, the application uses the **post** statement to trigger the BidUpdated event. This notifies all the end users interested in the same painting that the bid has been raised.

Once you post a custom event, it is treated exactly like any other event.

Responding to Events

In TOOL code, you define responses to events by using **event** statements. With the **event** statement, you register the event(s) that you want to respond to at this point in the code and the action to perform when a particular event occurs. For example, the Display method for a window typically registers to receive events on the various widgets on the window. The following method uses an **event loop** statement to process events from a window:

Example:
event loop statement

```
self.Open();
-- Open the window
event loop
  when task.Shutdown do
    exit;
  when <save_button>.Click do
    self.Save();
end event;
self.Close();
-- Close the window
```

Event registration

When the **event** statement is executed, Forte registers all the events listed in the statement and then waits to receive an event. Registering an event means notifying the object that will be posting the event that the current object is prepared to handle the event. This ensures that the current object will be notified when the event is actually posted.

When one of the registered events is posted and the **event** statement receives it, Forte then executes the corresponding **when** clause. When the **event** statement completes, the events are no longer registered. The registered events are the only events that can produce a response; other events are simply ignored.

Named event handlers

Besides listing events directly in the **event** statement, you can use the **register** statement to include a named event handler within the **event** statement. As described in *A Guide to the Forte 4GL Workshops*, a named event handler is a named block of TOOL code that provides programming to be executed in response to one or more events. The event handler provides reusable, modular event handling code that you can include in any number of **event** statements.

When you include a named event handler in an **event** statement, Forte registers all the events specified in the event handler. The following example illustrates the use of the **register** statement within the **event loop** statement.

Example:
register statement

```
event loop
  preregister
    -- Include the ArtObjectWindow’s event handler in this
    -- event loop.
    register artObjectWindow.artObjectHandler
        (artType = ’Performance’);
  when task.Shutdown do
    exit;
```

See NestedWindow example

Project: NestedWindow • **Class:** SellWindow • **Method:** Display

To cancel registration of the event handler before the **event** statement completes execution, use the DeregisterHandler method on the EventRegistration class.

The **event** statement has two variations: **event loop** and **event case**. Both variations register any number of events.

event loop statement

The **event loop** statement waits for any number of events and continues responding to them until you explicitly exit the loop. Event loops are useful for monitoring a continuous process, and responding to any number of events in whatever order they occur. The typical example is the event loop for the window, which continues responding to events until the end user closes the window.

event case statement

The **event case** statement responds to the first event it receives and then automatically exits the statement. An **event case** statement is useful for synchronizing with an independent process, when you need to respond once to whichever event occurs first. For example, when you start a new task (described under “Multitasking” on page 29), you can use an **event case** statement to wait for the event that indicates the task has completed.

Processing the event

Both **event** statements provide a statement block for each registered event, which is the code that processes the event. The statement block can include any TOOL statements. In the previous example, the code for the Click event on the Save button invokes a Save method.

Nested Events

You can nest **event** statements in two ways. First, the **when** clause for an event can contain an **event** statement. For example, in response to an event, you might use the **start task** statement to start a new task and then use the **event** statement to wait for the return event.

Example 1:
nested event statements

```
when <print_button>.Click do
  tsk : TaskDesc;
  tsk = start task printer.PrintIt()
    where completion = event;
    -- Return event
  event case
  -- Either complete or cancel
    when printer.PrintIt_completion do
      ... message for success ...
    when <cancel_button>.Click do
      tsk.SetCancel();
      -- Cancel the printer task
  end event;
```

Second, the code for an event can invoke a method that contains an **event** statement. This typically happens when you want to respond to an event by opening a new window and the Display method for the new window contains an **event** statement.

Example 2:
nested event statements

```
when <AddPaintingButton>.Click do
  add_painting_window : AddPaintingWindow = new;
  add_painting_window.Display
    (auctionMgr = self.AuctionMgr);
...
-- Display method for the AddPaintingWindow
self.Open(isAppModal = TRUE);
event loop
  when task.Shutdown do
    exit;
  ... More events...
end event;
self.Close();
```

When you nest **event** statements, the event queue will contain events registered by more than one statement. This affects the interactions between the various **event** statements. See "Event" on page 106 for information.

The Event Queue

The event queue is a series of registered events that are waiting to be processed by an **event** statement. Every time a registered event is posted, Forte adds the event to the end of the queue. Then each time the **event** statement is ready to handle an event, it uses the event at the top of the queue.

Queue order

The event queue order can not be guaranteed to be the same as the order in which events are posted. You cannot prioritize events in the queue, nor can you access or manipulate the event queue, with one exception—you can purge it using the PurgeEvents method on the Window class.

Multitasking and events

When you use multitasking, each task has its own event queue. Every time an event is posted, Forte checks all current tasks to see if the event is registered. If it is, Forte adds the event to the event queue for the task. This means that more than one task can respond to the same event. See "Event" on page 106 for information about tasks and event processing.

Posting Events

Within your method, you can explicitly trigger an event by using the **post** statement. The **post** statement allows you to trigger an event of any type and to specify values for the event's parameters.

This is especially useful for communicating between tasks (described under "Multitasking" on page 29) and for synchronizing independent processes. A good example is keeping concurrent windows up to date. In the following example, the AddBid method in the AuctionMgr class posts an event indicating that a painting has been added. Executing windows represented by the PaintingListWindow class monitor the event and add the new painting to their displayed list:

Example: posting events

```
-- ... in the AuctionMgr AddBid method ...
b : Bid = new;
b.SetValues (paintingForBid = paintingToAdd,
    currentBid = startingBid);
self.PaintingUnderBid.AppendRow(b);
-- Add to array
post self.PaintingAdded(painting = paintingToAdd);
 ...
--... in the PaintingListWindow Display method ...

event loop
...
  when self.AuctionMgr.PaintingAdded(addedPainting =
    painting) do
    -- A row from the array is to be added. Add to end.
    self.PaintingData.AppendRow(object = addedPainting);
  ... More events...
end event;
...
```

One advantage of using an event for communication (rather than invoking a method) is that the event is simply a signal that something interesting has happened. If the event is currently registered, the appropriate **event** statement will respond to it. If the event is not registered, it is simply ignored. (Of course, the disadvantage of using events is that you cannot assume that the event has been received.) Another advantage is that a single **post** statement notifies the entire application or all the other users of the application that the event has occurred.

Multitasking

Normally, the methods in an application are processed synchronously. When one method invokes another method, control passes to the invoked method. When the invoked method completes, control returns to the invoking method. Processing continues with the next statement.

Forte provides the option of asynchronous processing. The **start task** statement allows you to invoke a method asynchronously. The asynchronous method (or “task”) executes in parallel with the method that invoked it. Forte provides one statement for asynchronous processing:

Statement	Description
start task	Invokes a method asynchronously, which initiates a new task.

In a distributed system, using multitasking lets you to maximize your resources by simultaneously executing different methods on different machines. This is also useful for providing independent windows, where the end user can work on different tasks at the same time.

An effective way to structure an application is to create a master task that monitors several other tasks as they process by waiting for events (see “Communicating between Tasks” on page 32).

Multitasking and multithreading

In Forte, multitasking is implemented as multithreading whenever possible. If multithreading is not available on a particular platform, Forte simulates it. In either case, Forte takes care of the necessary synchronization.

What is a Task?

In Forte, every method runs as part of a task. The starting method for the application starts the “main” task by creating an object of the starting class type and invoking the starting method on that object. You can start subsequent tasks using the **start task** statement.

When you use the **start task** statement to invoke an asynchronous method, Forte starts a new task. A task is an individual thread of execution that runs to completion independently of any other tasks in the application. An asynchronous method does not return to the method that started it.

In the following example, a new window is opened as a separate task:

Example: opening a window as a new task

```
-- ... in Display method for the ListPaintingsWindow
  when <ViewPaintingButton>.Click do
  ...
    view_window : ViewPaintingWindow = new;
    start task view_window.Display(paintingName = name);
  ...
```

See Auction example

Project: Auction • **Class:** PaintingListWindow • **Method:** Display

A single task can invoke any number of synchronous methods. These are all considered to be part of the task. In the previous example, the asynchronous method that is invoked by the ListPaintingsWindow contains synchronous invocations of a number of methods.

Example: synchronous methods

```
-- ... in the Display method for the ViewPaintingWindow
self.UserName.SetValue(user_name);
self.theBid = self.AuctionManager.GetBidForPainting
  (name = painting_to_view.Name,
  -- output parameters
  paintingForBid = self.thePaintingForBid,
  currentBid = self.theCurrentBid,
  lastBidTime = self.theLastBidTime,
  lastBidder = self.theLastBidder,
  bidInProgress = self.theBidInProgress);
...
```

Task information

Every task is associated with a TaskHandle object, which provides access to information about the task. This is described under "Using TaskHandle and TaskDesc Objects" on page 33.

Event queue for task

Each task has its own event queue. Every time an event is triggered, Forte delivers the event to all tasks that have registered for it by adding the event to the task's event queue. This means that more than one task can respond to the same event. This also allows you to use events to communicate between tasks (see "Communicating between Tasks" on page 32).

Starting a New Task

To begin a new task, you use the **start task** statement, which invokes a method asynchronously. Use the method's parameters to pass information to the new task. The following example shows the Display method for the ListPaintingsWindow starting the Display method for the AddPaintingWindow as a separate task:

Example:
start task statement

```
-- ... in the Display method for the ListPaintingsWindow
...
  when <AuctioneerOnly.AddPaintingButton>.Click do
    add_painting_window : AddPaintingWindow = new;
    start task add_painting_window.Display
      (auctionMgr = self.AuctionMgr);
...
```

You can use the **start task** statement for any method. The method does not have to be specially designed for asynchronous processing. If the method contains a **return** statement, Forte simply terminates the method and the task when it reaches the return statement.

Tasks and transactions

By default if you use the **start task** statement within a transaction, the asynchronous method will not be part of the transaction. However, options for the **start task** statement allow you to run the task either as a dependent participant in the transaction that started it or as an independent transaction. See "Transactions" on page 34 for information on this.

Return event and exception event

Normally, the method that invokes a new task cannot tell when the asynchronous method completes. However, if the asynchronous method was defined with return and exception events (also called completion events), you can request notification when it completes successfully or terminates due to an exception. When you request notification, the asynchronous method posts a return event if it completes successfully and an exception event when it terminates due to an exception.

The return event is useful for synchronizing tasks. A single "parent" task can start several different "child" tasks, using an **event** statement to wait for notification that they have all completed. (This operation is sometimes called a *rendezvous*.)

Requesting the return and exception events automatically registers the events for the calling task. When the asynchronous method completes or terminates, Forte adds the appropriate event to the calling task's event queue.

This registration is unlike the event registration for the **event** statement. In the **event** statement, the event is registered just before the **event** statement is ready to process the event. In the **start task** statement, the return and exception events are registered when the task is started, even though your application will not begin to wait for those events until much later. Therefore, only the “parent” task that executes the start task statement is registered for the completion event of the started task.

See “Completing a Task” on page 31 for information on handling these events.

TaskDesc object

Another way you can communicate with the task is through the TaskDesc object. The **start task** statement always returns an object of the TaskDesc class. This object lets you access the task that you have started while it is still executing. For example, you can invoke the SetCancel method on the TaskDesc object to cancel the new task before it completes. See “Using TaskHandle and TaskDesc Objects” on page 33 for more information.

Completing a Task

A method that is invoked asynchronously runs independently to completion. When the method completes, the task terminates. When the application’s “main” task completes, the application terminates.

Forte executes the method to completion or to the **return** statement. The only time Forte terminates the task prematurely is when there is an exception that the task cannot handle (see “Exception Handling” on page 42 for information on exceptions).

Requesting return and exception events

Normally, the method invoked by the **start task** statement does not notify the invoking task that it has completed or that it was terminated. However, if the method has been defined with return and exception events, you can use the **completion event** option of the **start task** statement to request these events.

Return event

Forte automatically posts a return event to the invoking task when the started task completes. The parameters for this event are the input-output and output parameters defined for the method. These parameters have whatever values are current when the method completes.

Return parameter

If the method has a return type, the last parameter for the return event is a parameter called “return.” This has the return value specified by the method’s **return** statement.

Handling the return event

When you use the **event** statement to handle the return event, you can use these parameters for further processing. In the following example, a modification to the ViewPaintingWindow Display method shows how a return event would work.

Example:
handling the return event

```
-- ... in the ViewPaintingWindow Display method...
start task self.ImageManager.GetImage
  (name = self.thePaintingForBid.Name)
  where completion = event;
event loop
  when self.ImageManager.GetImage_return
    (theImage = return) do
    self.PaintingImage = theImage;
  when self.ImageManager.GetImage_exception
    (e = exception, errStack = errMgr) do
    -- the exception message was suppressed
    ImageStatusMessage.SetValue(‘Image Not Found.’);
    <ImageStatusMessage>.State = FS_VISIBLE;
...
end event;
```

Exception event

Forte automatically posts the exception event to the calling “parent” task when the started task is terminated due to an exception. For information about exceptions in general, see “Exception Handling” on page 42. For information about exception event handling, see “Start Task” on page 147.

Transactions and completion

When an asynchronous method completes normally, this is considered successful completion. However, if the task associated with the method was participating in another task's transaction and the method (and task) is terminated through an exception, this is considered a failure. Failure aborts the transaction in which the task is participating.

Distributed task completion

The execution of a task moves between partitions whenever a method is invoked on a reference to a remote object. As with local execution, the method invocation may be synchronous or asynchronous. If the method invocation is synchronous, the task is blocked in the calling partition while it executes in the remote partition. If the method invocation is asynchronous, the invoking task may continue executing while the subtask executes concurrently in the remote partition.

Forte provides methods to terminate active tasks running locally or remotely. The PostShutdown and SetCancel methods on the TaskDesc and TaskHandle classes (described in the manual Framework Library online Help) allow you to asynchronously terminate a single task and/or its subtasks. Also, the settings of the Window class SystemClosePolicies (described in the manual Display Library online Help) cause the PostShutdown method to be invoked on related tasks.

Remote task shutdown differs from local task shutdown

Forte applies task shutdown policies to all local members of a task. If all tasks are running locally the tasks are notified of the termination request exactly as described above. Because Forte applies the shutdown policies differently to tasks or subtasks that are executing in a remote partition, you may need to take additional steps to terminate remote tasks/subtasks.

Terminating tasks running remotely

For example, if you locally terminate a task that is currently synchronously executing a method in a remote partition, the remote execution of the task will run to completion unless you explicitly terminate the task in *that* partition (you may do this by invoking the PostShutdown or SetCancel methods in the remote partition). In order to terminate remote subtasks started by asynchronously invoking a method on a remote object, you must explicitly invoke the PostShutdown or SetCancel method on the TaskDesc or TaskHandle associated with that task.

Communicating between Tasks

The return event enables you to receive notification when an asynchronous method completes. Tasks can also communicate using the **post** and **event** statements to signal an event's occurrence and respond to an event. Event parameters also pass information regarding an event between tasks. Thus, events and event parameters allow you to synchronize concurrent windows as well as pass status information between tasks.

For example, you may wish to monitor a task while it is executing. You can do this by using the **event** statement in one task to wait for events being posted by another task. The task being monitored can then use the **post** statement to trigger events when relevant changes occur on an object. In the following example, the AddBid method of the AuctionMgr class can notify any tasks that are executing the Display method of the PaintingListWindow class that a painting has been added:

Example: communicating between tasks

```
-- ... in the AuctionMgr AddBid method ...
b : Bid = new;
b.SetValues (paintingForBid = paintingToAdd,
    currentBid = startingBid);
self.PaintingUnderBid.AppendRow(b);
-- Add to array
post self.PaintingAdded(painting = paintingToAdd);
 ...
--... in the PaintingListWindow Display method ..
event loop
```

```
...
  when self.AuctionMgr.PaintingAdded(addedPainting = painting) do
    -- A row from the array is to be added. Add to end.
    self.PaintingData.AppendRow(object = addedPainting);
... More events...
end event;
...
```

Shared Objects

A shared object is an object that will be concurrently accessed by multiple tasks and requires that this access be regulated. You create a shared object by defining the class as Shared in the Project Workshop, and setting the object's IsShared attribute to TRUE (explicitly, or by setting the default value for the class to TRUE).

TOOL provides a mutex locking mechanism for shared objects to prevent conflicts when multiple tasks try to access or change the object's state (this mechanism is described in the Framework Library online Help manual under the Mutex class). If one task modifies a shared object's attribute, TOOL locks the object until the change is complete. If one task invokes a method on a shared object, TOOL does not allow another task to execute a method on the object until the first task completes its method. Other tasks attempting to invoke methods on or access/modify attributes of the same object are "blocked." Once the first task completes the method, another task is allowed to continue. Note, however, that the object is not locked while waiting for an event during the processing of an **event** statement.

Multiple tasks should not operate concurrently on non-shared objects (objects whose IsShared attribute is not set to TRUE). If two tasks do operate on and access the same non-shared object, the results are unpredictable.

See the *Forte 4GL Programming Guide* for more information on shared objects.

Using TaskHandle and TaskDesc Objects

TaskHandle class

Every task is associated with an object of the TaskHandle class. This class provides information about the current task and provides access to the Forte runtime system. The ErrorMgr attribute provides access to the error manager for the task and the Part attribute provides access to the partition for the task. See the TaskHandle class in the *Framework Library online Help* for further information about using these attributes and the TaskHandle class.

Task key word

To make it easy to reference the current task, TOOL provides a special **task** key word. Use the **task** key word to reference the TaskHandle object associated with the current task. The following example shows using the task key word to access the current error stack:

```
-- Clears the error stack after printing
task.ErrorMgr.ShowErrors(clearout = TRUE);
```

TaskDesc class

As described above, the **start task** statement returns an object of the TaskDesc class. This object provides the calling task with access to the new task that it has started. For example, the SetCancel method of the TaskDesc class lets you cancel the new task while it is currently executing. See the TaskDesc class in the *Framework Library online Help* for details.

Transactions

A Forte transaction is one or more statements that execute as a single unit of work. A transaction is either successful or unsuccessful; either all the statements or none of them take effect. If you are unfamiliar with the basic concepts of transaction management, there are many texts you can read on this topic.

Start a transaction with TOOL or a Framework class

TOOL provides the following statements for transaction management:

Statement	Description
begin transaction	Specifies a statement block that is treated as a single transaction.
start task	Lets you start a new task as an independent or dependent transaction.

More sophisticated transaction management is available through the Forte TransactionDesc and TransactionHandle classes described in the Framework Library online Help.

The **begin transaction** statement starts a transaction statement block; the **end transaction** clause of the statement denotes the end of the transaction. All the statements in the block and all methods invoked from the block are treated as a single transaction, including SQL statements participating in a database transaction.

Successful transaction

When the statement block executes successfully, Forte automatically commits the transaction, causing all the statements in the transaction to take effect.

Unsuccessful transaction

When an exception is raised, Forte aborts the transaction, and control passes either to the **transaction** statement's exception handler or, when an exception is raised that closes the transaction block, to the closest exception handler. None of the statements in the transaction take effect. (A notable exception to this is that changes made to variables and non-transactional objects are not part of the transaction and therefore cannot be rolled back. See "Transactional and Shared Objects" below for more information.)

The following sections provide more information about using transactions.

Transaction Types

Transactions can be independent, dependent, or nested. These three types determine how a transaction enclosed in another transaction affects the enclosing transaction. See "Transaction Type" on page 96 for more information.

Transactional and Shared Objects

Transaction and non-transactional objects

A *transactional object* is an object that can participate in a transaction. Changes made to the object during a transaction are logged so the changes can be rolled back if the transaction is aborted. Outside the context of a transaction, a transactional object behaves exactly like any other object. You create a transactional object by defining the class as Transactional in the Project Workshop, and setting the object's IsTransactional attribute to TRUE (explicitly or by setting the default value for the class to TRUE).

A *non-transactional object* is an object that is not affected by the success or failure of a transaction. Changes made to a non-transactional object during a transaction are *not* rolled back. You can make any object non-transactional by setting its IsTransactional attribute to FALSE, or by setting the default value for the class to FALSE.

Transaction locks

A transactional object can also be shared. You indicate that it is shared by setting both its IsTransactional and IsShared attributes to TRUE. To access or modify a shared transactional object (either through a public attribute or a method), the transaction must acquire a transaction lock on the object. The transaction lock is in addition to the normal mutex lock that regulates concurrent shared object access. Once a transaction acquires a lock on an object, it holds the lock until the transaction ends (either by aborting or committing it). When the transaction ends, the lock is released and the next waiting transaction (if any) is granted access.

See the *Forte 4GL Programming Guide* for more information on shared and transactional objects.

Forte Distributed Transactions

Because a Forte application is distributed, an executing transaction is also likely to be distributed (a transaction can be local, if it never leaves the partition in which it started). For example, if a method that displays a window is running as a transaction, the work that displays information on the client is part of the same transaction as the work that gets the information to be displayed by selecting it from a database located on the server. To implement distributed transactions, Forte uses a simplified two-phase commit protocol for all Forte objects.

Committing a successful distributed transaction

A distributed transaction can access more than one database or more than one database resource manager. Forte coordinates all commits and aborts. For example, if a transaction accesses more than one database resource manager, and one of the resource managers aborts the transaction, Forte notifies the other resource managers that the transaction is aborted. Likewise, when the transaction commits, Forte signals all resource managers to commit.

Refer to the manual *Accessing Databases* for a more detailed discussion of issues related to two-phase commit and failure during commit of a distributed transaction.

Transactions and Multitasking

As mentioned under “Multitasking” on page 29, you can start a new task as a transaction. The new task can either start an independent transaction or it can participate in an existing transaction.

Because each task can begin its own independent transaction, any number of independent transactions can execute concurrently. For example, if you have two concurrent windows, each running as an independent transaction, each transaction may execute work on the client as well as work on one or more servers.

Multiple tasks can also participate in a single transaction. One task can begin a transaction and, within that transaction, start multiple tasks as “dependent” participants in the transaction. You can do this by using the **dependent** or **nested** option on the **start task** statement or by using the Join method on the TransactionHandle class.

If a dependent participant’s transaction is aborted or the task fails, the caller’s transaction is also aborted. Likewise, the caller’s transaction cannot be committed until all the dependent transactions have committed. If the task that originally began the transaction attempts to commit while other tasks are still participating as dependents of the transaction, the committing task will wait until the participants have all completed their work.

An extension of the **dependent** option is the **nested** option. When you use the **start task** statement with the **nested** option, the new task immediately starts a nested transaction. This is equivalent to being a dependent participant, except that if the new task fails, it will only abort the nested transaction without affecting the caller’s transaction.

Note that if you plan to use the same database session in multiple concurrent tasks, all the tasks must be in the same transaction.

See “Start Task” on page 147 for information about the **start task** statement. See the *Framework Library online Help* for information about the TransactionHandle class.

Using TransactionHandle Objects

transaction key word

Every transaction is associated with an object of the TransactionHandle class. This class provides information about the current transaction and methods for managing it. You can use the special TOOL key word **transaction** to reference the current transaction (actually, the current TransactionHandle object). The following example shows a transaction block that can abort:

Example: use of TransactionHandle object

```
begin transaction
  sql insert into ...
  sql update ...
  if (condition_that_should_abort) then
    transaction.Abort(raiseException = TRUE);
  end if;
  ... more statements in the transaction ...
exception
-- The transaction has been aborted
  when e : GenericException do
    task.ErrMgr.ShowErrors(TRUE);
-- Print errors
end transaction;
```

(A TransactionHandle object is available even when there is no current transaction. In this case, the handle is considered “idle.”)

As well as duplicating all the functionality of the **begin transaction** statement, the TransactionHandle class provides additional methods for managing transactions. These methods allow you to:

- Start a transaction in one method and end it in another method, or start and end in different code blocks.
- Make dynamic decisions about the type of transaction used at a certain point.
- Make a dynamic decision as to the extent of the participation of a task in a transaction (if you are using events to coordinate a transaction participating across tasks). A task participating in the transaction can abort or commit the transaction at any point.
- Get diagnostic information about a particular transaction.

See the TransactionHandle class in the manual *Framework Library online Help* for more information.

Be careful combining statements and methods

Although you can combine the TransactionHandle methods with the **begin transaction** statement, it is important to do so carefully. The **begin transaction** statement itself invokes TransactionHandle methods, and because these methods are invoked in a particular sequence, you can disrupt this sequence if you use the methods incorrectly. Some examples follow:

- If you start and end a nested transaction with BeginNested and CommitNested methods within a **begin transaction** statement block, this will work correctly.
- However, if you include a transaction.Commit method within a **begin transaction** statement block to commit the current transaction, this will cause an exception when Forte reaches the end of the **begin transaction** statement. This is because Forte will try to commit the same transaction again.

- Likewise, if you use the BeginNested method within the **begin transaction** statement block without the corresponding CommitNested method, this would cause the transaction to remain active after reaching the end of the **begin transaction** statement.
- If you use the variation of the Abort method that raises an exception within the **begin transaction** statement block, control will pass correctly to the **begin transaction** statement's exception handler. However, if you use the variation of the Abort method that does not raise an exception, the application will be in an invalid state because the remaining statements within the **begin transaction** block will be executed outside of a transaction. For example:

```
begin transaction
...
transaction.Abort(FALSE); -- No exception
...
-- At this point you are no longer in a transaction
exception
...
end transaction;          -- Error: not in transaction
```

This would also occur if you were to invoke the version of the AbortNested method that does not raise an exception from within a **begin nested** transaction. For example:

```
begin transaction
...
begin nested transaction
...
transaction.AbortNested(FALSE);   -- No exception
...
-- At this point you're no longer in a nested transaction
exception
...
end transaction;                -- commit outer transaction!
-- At this point you are no longer in a transaction
exception
...
end transaction;                 -- error that not in transaction
```

In both examples, the Abort and AbortNested methods should pass a value of TRUE for the raiseException parameter in order to keep the runtime state of the transaction synchronized with the lexical scope of the **begin transaction** statement.

Interacting with a Database

To interact with a database from a Forte application, you first identify the databases that you wish to work with and you start one or more database sessions for each database. Then, in the TOOL SQL statements, you specify which database session to use. The manual *Accessing Databases* describes using database sessions.

To manipulate database data, you use the TOOL SQL Data Manipulation Language (DML) statements and the Forte classes in the library named GenericDBMS.

The following table lists the TOOL SQL statements:

SQL Statement	Description
sql select	Retrieves a single row from a database table, or, if you are selecting into an array, retrieves multiple rows.
sql insert	Adds a row to a database table.
sql delete	Removes rows from a database table.
sql update	Replaces values in a database table.
sql open cursor	Selects rows from database tables for use with a cursor.
sql close cursor	Closes a cursor.
sql fetch cursor	Retrieves a row from a cursor and stores the values in Forte variables.
sql execute immediate	Executes a single SQL statement specified as a literal string, a string variable, or a TextData variable.
sql execute procedure	Executes a database procedure, which only return output parameters and a single return value from the procedure.

Using standard SQL guarantees that your application will run on any of the database management systems that Forte supports. While Forte allows you to use most vendor-specific extensions to SQL, if you do so, your code may not be portable across different database management systems.

The Forte GenericDBMS library classes provide the ability to execute dynamic SQL. These classes provide corresponding methods for all the TOOL SQL statements.

The following sections provide further information about working with the SQL statements. For information on the database management classes, see the manual *Accessing Databases*.

Selecting Rows

You can select a single row or a set of rows from a database by using the **sql select** statement. You can select a single row from a database using the **sql execute procedure** statement. If you want to step through a set of rows, you must either use a cursor (described below) or use the **for** statement (described under "For" on page 118).

sql select statement

The **sql select** statement allows you to retrieve a row or a set of rows from a database table, and to store the values in Forte variables or attributes. You specify the conditions that the row must meet and list the Forte variables and/or attributes to store the result values.

sql execute procedure statement

The **sql execute procedure** statement executes a database procedure using the parameter values you specify. One way you can use this statement is to execute a procedure that selects one row from the database. If your DBMS allows it, the output parameters for the procedure let you return the database values from the procedure to your Forte application. If not, you can specify a list of Forte variables or attributes to contain the values.

Using Cursors

A cursor is a row marker that you can use for working with a set of rows from a database. Cursors enable you to step through the set of rows one row at a time.

To define a cursor, you must use the Cursor Workshop (see *A Guide to the Forte 4GL Workshops* for information about how to do this). Because the cursor definition is associated with the project as a whole, you can reference it from more than one method.

After defining a cursor, there are two different ways you can use it. You can use the **for** statement to repeat a statement block for each row in the result set of the cursor. Or you can use the **sql open cursor, sql fetch cursor**, and **sql close cursor** statements to step through the cursor's result set one row at a time. If your particular database system allows it, you can also use a cursor with the **sql update where current of** or **sql delete where current of** statements to update or delete the row to which the cursor is pointing.

for statement with a cursor

To use the **for** statement with a cursor, you simply use the cursor name. The **for** statement automatically opens the cursor, fetches the rows one at a time as it goes through the loop, and then closes the cursor.

Fetching rows

To use the **sql fetch cursor** statement, you must begin by opening the cursor with the **sql open cursor** statement. You can then use the **fetch** statement to retrieve one row at a time. Finish by using the **sql close cursor** to close the cursor.

sql open cursor statement

The **sql open cursor** statement executes the **select** statement associated with the cursor and positions the cursor before the first row in the result set. At this point, you specify the values for any placeholders used in the original cursor declaration.

sql fetch cursor statement

The **sql fetch cursor** statement lets you move one row at time through the result set of the cursor's **select** statement. This statement does not have to be in the same method as the **sql open cursor** statement. After you open the cursor, the cursor is positioned before the first row in the result set. The first time you use the **sql fetch** statement, Forte moves the cursor to the first row in the result set. You can then perform any processing you wish on the individual row. With each successive **sql fetch** statement, Forte moves the cursor forward one row. You can continue using **sql fetch** to move through the result set until you reach the last row in the result set.

sql close cursor statement

The **sql close cursor** statement lets you close the cursor. This statement does not have to be in the same method as the **sql open cursor** or **sql fetch** statements. After the cursor is closed, it cannot be used again until you use another **sql open cursor** statement to open it.

Updating the Database

TOOL provides three statements for updating database data: **sql insert**, **sql update**, and **sql delete**.

sql insert statement

The **sql insert** statement adds a new row to the specified table. You can either provide a list of values to use for the new row or you can use a **select** statement to get the values from another table.

sql update statement

The **sql update** statement replaces the current column values in the selected row or rows with the new values that you specify. You use the **set** clause to specify the new values for the columns. You use the **where** clause to select the rows that you wish to update. Without the **where** clause, the **update** statement changes the values in all the rows in the table.

sql delete statement

The **sql delete** statement removes the specified rows from a database table. The **where** clause identifies the particular rows to be deleted. If you do not include the **where** clause, Forte deletes all the rows in the table.

Vendor-Specific Extensions

You can use almost all vendor-specific extensions to the ANSI standard SQL syntax that are allowed by your database management system. However, if you use vendor-specific extensions, your code is no longer generic but is valid only for the particular database systems that support that syntax.

The following examples illustrate using vendor-extensions for three different vendors:

Example: vendor extensions to ANSI SQL

```
-- SYBASE compute.
sql select type, price from titles
  where type like "%cook"
  order by type, price
  compute sum(price) by type;
```

```
-- RDB cast.
sql select * from alltypes
  where cast(col06 as double precision) > 10.0;
```

```
-- ORACLE outer join.
sql select anything(a, character), b, c from aTable, atableToo
  where
  aTable.x (+) = :aVariable
  and y = :aVariable order by foo asc;
```

Note Because vendor-specific syntax is passed directly to the DBMS, Forte does not detect syntax errors in vendor-specific clauses.

Forte Transactions and Database Transactions

This section describes the use of explicit Forte transactions when interacting with a database. For a more thorough explanation of the interaction between Forte transactions and database transactions, refer to the manual *Accessing Databases*.

Use explicit Forte transactions

For best performance, you should place every SQL interaction with a database in an explicit Forte transaction. Typically you start a Forte transaction using the TOOL statement **begin transaction**; see "Transactions" on page 34 for information about defining Forte transactions. Then, you include SQL in a Forte transaction using either TOOL SQL statements (such as **sql select**) or the equivalent methods (such as the Select method) on the DBSession class.

When you include SQL in an explicit Forte transaction, Forte automatically starts a database transaction. This database transaction must commit in order for the enclosing Forte transaction to commit. If the database transaction aborts, the enclosing Forte transaction aborts. Or, if the enclosing Forte transaction aborts, Forte rolls back the database transaction. If you do not enclose the SQL statements in a Forte transaction, Forte executes each individual SQL statement as an implicit transaction (described under "Implicit Forte Transactions" on page 41).

Do not use SQL transaction management

Because Forte coordinates the entire transaction, you should not use the **sql execute immediate** statement in the Forte transaction to execute any SQL statement that affects the DBMS transaction (such as rollback, commit, or setautocommit). If you do, your DBSession object's transactional state will become out of sync; this will most likely cause an exception which will cause the Forte transaction to fail.

You must also exclude any SQL statements that your DBMS does not allow within a transaction (for example, some systems do not allow DDL statements in transactions).

Transactions and multiple database sessions

A single Forte transaction can access more than one database session, more than one database, or more than one database resource manager. Forte coordinates all commits and aborts. For example, if a transaction accesses more than one database resource manager and one of the resource managers aborts the transaction, Forte notifies the other resource managers that the transaction is aborted. Likewise, when the transaction commits, Forte signals all DBMS resource managers to commit the transaction.

Refer to the manual *Accessing Databases* for a more detailed discussion of issues related to two-phase commit and failure during commit of a distributed transaction.

Multitasking and database sessions

Note that if you plan to use the same database session in multiple concurrent tasks, all the tasks must be in the same transaction. You can do this by using the **dependent** option on the **start task** statement or by using the Join method on the TransactionHandle class. See "Transactions and Multitasking" on page 35 for information on transactions and multitasking.

Implicit Forte Transactions

If you issue a single SQL statement outside of an explicit Forte transaction, an implicit Forte transaction begins. This is true for most SQL statements, including **sql select**, **sql open cursor**, and **sql execute immediate**; it is not true for **sql fetch cursor** or **sql close cursor**. The implicit transaction contains only the SQL statement; it starts before the SQL statement executes and is committed after the statement completes. No other TOOL statements are affected.

Explicit transactions perform better

Although implicit transactions ensure the integrity of your data with respect to the database, they are slower and consume more memory than explicit transactions. Execution of multiple SQL statements enclosed in separate transactions is generally slower than if the statements are executed within a single transaction.

Use explicit transaction for queries

You will get better performance if you put queries (**sql select** statements and **sql open cursor** followed by **sql fetch statements**) in explicit Forte transactions. For cursors, the result sets must be buffered, which consumes memory and might cause performance to deteriorate because Forte retrieves the entire result set before executing the next statement.

Exception Handling

Forte reports all errors as exceptions and provides statements that you can use to handle the exceptions in your code. Forte also lets you use the same mechanism for reporting and handling your own errors.

Exception handling means passing an “exception” to a special exception handler, outside the current block of code. The exception handler provides the code that handles the exception.

You use the following TOOL statements for exception handling:

Statement	Description
exception	Provides exception handing for the current statement block or compound statement.
raise	Generates an exception to be handled by an **exception** handler.
begin	Defines a compound statement, which provides local exception handling for a group of statements.

The advantages of using exception handling over traditional error handling are:

- Exception handling isolates your special-case code so that the body of the method is cleaner and more straightforward. You do not have to check for errors every time you use a statement.
- Your code cannot ignore errors because unhandled exceptions will terminate the task.
- The error manager for the task makes it easy to print out and work with error messages.

The following example illustrates exception handling:

Example:
exception handling

```
begin transaction
  sql insert into ...
  sql update ...
  if (condition_that_should_abort) then
    transaction.Abort(raiseException = TRUE);
  end if;
  ... more statements in the transaction ...
exception
  -- The transaction has been aborted
  when e : AbortException do
    -- Display error but continue
    task.ErrMgr.ShowErrors(TRUE);
  when e : GenericException do
    -- Get out
    raise;
end transaction;
```

About Exceptions

Exception classes

An exception is a signal that an abnormal condition occurred. Forte provides a special set of classes for exceptions, the GenericException class and it subclasses. The class of an exception determines the type of exception. The exception's attributes contain information that the exception handler can use when responding to the exception.

The Forte exceptions fall into the following six categories:

Exception	Description
AbortException	Forte raises this type of exception when it is aborting a transaction. Standard recovery is to retry the transaction.
ResourceException	Forte raises this type of exception when there is an error from a resource manager, such as a DBMS, file manager, or lock manager. You can often recover from this type of exception.
CancelException	Forte raises this type of exception when one task cancels another task.
ArithmeticException	Forte raises this type of exception when there is overflow, underflow, or divide by zero in an arithmetic operation.
DataTypeException	Forte raises this type of exception when there is an incompatible attempt to convert one data type to another, such as "abc" to an integer or an invalid null assignment.
DefectException	Forte raises this type of exception when there is a bug in your software (or in the Forte software). Examples are a NIL object reference or incorrect usage of a method. In general, you shouldn't try to recover from these exceptions; you should fix the bug.

See the *Framework Library online Help* for details on the Forte exceptions.

User-defined exceptions

You can also create your own exception classes. In the Project Workshop, you can create a subclass of the Forte GenericException class. The GenericException class provides the attributes and methods necessary for working with the exception. However, because you can raise an object of any class as an exception, the exception classes that you define do not have to be subclasses of GenericException.

Exceptions are generated in two ways. First, under the appropriate conditions, Forte generates exceptions. An example is the AbortException, which Forte raises when it aborts a transaction. Second, you can generate exceptions explicitly by using the **raise** statement. You can raise an object of a Forte exception class or one of your own exception classes. This is described under "Raising Exceptions" on page 46.

When any exception is raised, Forte skips the remaining statements in the block and executes the closest **exception** clause. If the clause can handle the exception, Forte executes the appropriate exception handling code and then ends the statement block. Control resumes immediately after the **exception** clause.

If the closest **exception** clause cannot handle the exception, Forte tries the enclosing statement blocks until it finds an **exception** clause that can handle it. If the current method cannot handle the exception, Forte tries the invoking methods. If the task cannot handle the exception, the task terminates.

The following section provides more information about handling exceptions.

Handling Exceptions

Exception handler for statement

You always handle exceptions by using an **exception** clause. This clause is either associated with a statement or a compound statement. The syntax of a statement shows whether it has an exception handler.

Because the body of a method is a statement block, each method can have its own exception handler. In addition, all the individual TOOL statements that contain statement blocks can also have their own exception handlers.

Exception handler for compound statement

A compound statement is a subset of statements within a statement block that provides its own exception handling. You can use a compound statement anywhere that an individual statement is allowed. A compound statement starts with a **begin** statement and concludes with an **end** clause.

The following example illustrates checking for an error on connect in a compound statement:

Example: exception handler for compound statement

```
begin
-- Check for exception on connect to DB
  self.io.Write(‘Connecting to ‘);
  self.io.WriteLine(dbName);
  self.Session = dbMgr.ConnectDB (resourceName = dbName,
  userName = uname, userPassword = upassword);
  self.io.WriteLine (‘Connected successfully.’);
exception
  when e: GenericException do
    task.ErrMgr.ShowErrors(TRUE);
end;
--Check for exception on connect
```

The **exception** clause specifies the exception classes that it is prepared to handle. For each type of exception, the **exception** clause provides a statement block (in a **when** clause) that is executed when the exception is handled. The **else** clause specifies code that is executed for any exceptions that are not individually handled by the statement. See “Exception” on page 113 for information about multiple **when** clauses and when each clause is selected.

The statement block for an exception can include any TOOL statements. If you want to retry the method after handling the exception, you can include the method in a loop statement. Then, in the statement block for the exception, you can repeat the loop.

Example: repeating loop in exception clause

```
i : integer = 5;
while i > 0 do
  aborted : boolean = FALSE;
  begin transaction
    sql insert ...
    ... other statements in transaction ...
  exception
-- For the transaction block
    when e : AbortException do
      i = i - 1;
      aborted = TRUE;
      exit;
    else
-- Any other exceptions
      raise;
  end transaction;
  if not aborted then
    -- Exit the while block, as transaction is ok.
    exit;
  end if;
end while;
if i <= 0 then
  -- The transaction was aborted too many times.
  ...print message of warning to user...
end if;
```

The error manager

Every task is associated with an object of the ErrorMgr class, which provides an error stack that you can print or query. As errors occur in the runtime system, they are added to the error stack for the task. When Forte raises exceptions, these are also added to the stack. At any point while you are handling exceptions, you can print or display the messages in the stack with the ShowErrors method. In addition, when you request the exception event in the **start task** statement, the exception event contains an **errMgr** parameter. This parameter is an object of type ErrorMgr, which contains the error manager for the task. See the *Framework Library online Help* for information on the ErrorMgr class. See “Start Task” on page 147 for information about the exception event.

Handling AbortException and CancelException

You must be sure to handle AbortException and CancelException at the appropriate points in your application. The AbortException is intended to abort a transaction, and should therefore be handled in the exception clause of the **begin transaction** statement (not within its nested code). The CancelException is intended to cancel a task, and should therefore be handled in by the task’s outermost exception handler (not within its nested code). See “Exception Handling” on page 99 for information about exception handling for the **begin transaction** statement and “Invoking the Method” on page 148 for information about the invoked method of the **start task** statement.

Handling unexpected exceptions

In general, it is a good policy not to handle unexpected exceptions. You should only handle those specific exceptions that you are prepared to handle at the current point in the application. In particular, you should not handle AbortException and CancelException at incorrect points in the application. This could happen if you are providing generic code for unexpected exceptions.

However, there may be cases where you wish to provide general recovery for an unexpected exception and then raise it again to the enclosing exception handler. See “Else Clause” on page 115 for information about using the exception handler’s **else** clause correctly to handle unexpected exceptions.

Raising Exceptions

To generate an exception, you use the **raise** statement. The **raise** statement specifies one object to be the exception. You can raise an object of any class.

Example: raising an exception

```
-- A new UserException subclass of GenericException exists
userExc : UserException = new;
userExc.SetWithParams(severity = SP_ER_ERROR,
  message = ‘Error with <%1> on attempt number <%2>.’,
  param1 = TextData(value = ‘User Code’),
  param2 = IntegerData(value = n));
task.ErrMgr.AddError(userExc);
raise userExc;
```

When you raise an exception, it is treated like any other exception. It will be handled by the closest **exception** handler that is prepared to handle exceptions of that class (or any of its superclasses).

If you wish to add exceptions to the Forte error stack, the exceptions you raise must be subclasses of the GenericException class. You can then use the AddError method of the ErrorMgr class to add the exception to the error stack for the task. This keeps the stack up to date.

Error Handling

For cases when exception handling is inappropriate, Forte lets you use traditional error handling. You do this by using the **return** statement to return an error status to the invoking method. In the invoking method, you provide processing based on the error status. A typical solution is to assign the method’s return value to a variable and then use a conditional statement to determine how to respond to it.

Chapter 2

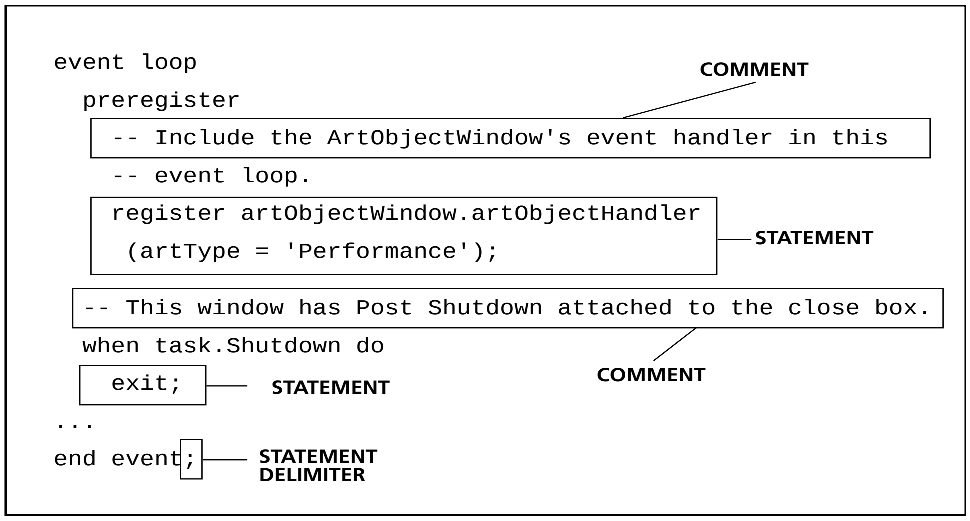
Language Elements

This chapter describes the basic language elements of TOOL, which include the following:

- statements
- comments
- names
- simple data types
- objects
- array objects
- variables
- named constants
- cursors
- service objects

TOOL Statements and Comments

Forte methods are composed of TOOL statements and comments. Figure 2 illustrates statements and comments:



```
event loop
  preregister
    -- Include the ArtObjectWindow's event handler in this
    -- event loop.
    register artObjectWindow.artObjectHandler
     (artType = 'Performance');
   -- This window has Post Shutdown attached to the close box.
  when task.Shutdown do
    exit;
...
end event;
```

Figure 2 Statements and Comments

Case insensitive

TOOL is not case sensitive. You can enter statements, comments, and other language elements in uppercase, lowercase, or any combination of the two.

Statements

Semicolon delimiter

A TOOL statement must end with a semicolon. You can start TOOL statements anywhere on a line. Although you can put multiple statements on one line, your code will be more readable if you begin each statement on a new line.

Newline characters within statements

You can use newline characters (line breaks) almost anywhere in TOOL code, but you should avoid line breaks in the middle of identifiers, constants, or operators. In fully qualified names (names that use dot notation), you can use line breaks before or after the period, as in the following example:

```
SqlRunServiceObjectReference(SoRef)
  .SqlRunTimeService
  .SqlSupportObj
  .UpdateArrayOfTargetTableBRows(targetTableArrayB);
```

Statement Blocks

Many TOOL statements contain *statement blocks*, or embedded statements. You can include any statement in a statement block. The syntax of a statement block is:

statement; [*statement*; **...**]

Statement blocks are significant because they determine the *scope* for any variables or constants that are declared. See "Name Resolution" on page 52 for further information about name scopes. A method always contains one statement block within the statements **begin** and **end**, although these two statements are automatically added by the Forte workshops, so you may never see them. The method statement block usually contains many more statement blocks. While a statement block is contained by a statement, the containing statement is not in the same statement block. The following example shows a method statement block with three statements (two of which include statement blocks).

1st statement

2nd statement

3rd statement

```
method myclass.mymethod
begin
-- A for loop to calculate factorial
j : integer = 1;
for i in 1 to 10 do
  j = j  * i;
  -- Put out a diagnostic message. ‘t’ is reinitialized
  -- each time through.
  t : TextData = new(value  = ‘Value is now: ’);
  t.Concat(j);
  task.Part.LogMgr.PutLine(t);
end for;
-- A transaction block
begin transaction
  sql insert into ...
  sql update ...
end transaction;
end method;
```

Compound statements

You can use the **begin** statement to nest a compound statement within a statement block. The compound statement determines the scope for any variables and constants that are declared between the **begin** and **end** clauses.

Example:
compound statement

```
-- Example of compound statement to read a text file.
while ...some condition... do
  outText : TextData = new(value = ‘FILE NOT READ.’);
  begin
-- Check for errors on file open
    f : File = new;
    fname : TextData = new;
    fname = ... prompt user for name ...;
    f.SetLocalName(name = fname);
    f.Open(accessMode = SP_AM_READ);
    f.ReadText(target = outText);
  exception
  -- Any errors in naming, opening, or reading file.
    when e : FileResourceException do
      task.ErrMgr.ShowErrors(TRUE);
  end;
  ...more of the while loop. outText is set...
end while;
```

See “Begin” on page 93 for information about compound statements.

Statement Labels

You can assign statement labels to identify certain TOOL statements. Statement labels allow you to direct program control to a specific statement. Labels are usually used for control statements, or statements that can effect the flow of control of a program. Examples of control statements are **case**, **event loop/case**, **if**, **for**, **while**, **continue**, **exit**, and **return**. However, not all control statements allow labels, and some non-control statements do allow labels. Chapter 3, “TOOL Statement Reference,” indicates which statements can have labels.

Statement labels can be any legal TOOL name (see “Names” on page 51). In the following example, the label “reprompt” is used with the **exit** statement to exit both the **for** loop and the **while** loop; if the label were omitted, the exit statement would leave only the **for** loop, but not the **while** loop.

Example:
statement label

```
reprompt : while (TRUE) do
  for i in 1 to 10 do
    ... something done 10 times ...
    if self.Window.QuestionDialog
        (‘Go again?’, BS_YESNO, BV_YES)  = BV_NO then
      exit reprompt; -- Leave the while, not just the for
    end if;
  end for;
  ...more in the while loop ...
end while;
```

Comments

You can include two types of comments in methods: single-line comments and block comments. Comments are ignored when the code is compiled and executed.

Single-Line Comments

Single-line comments begin with the characters “--” or “//” and end with the end of line character. The system ignores all characters between these two delimiters.

Example:
single-line comments

```
// Here is a comment
-- Here is another style comment.
x = 10;         // Here is one at end of line
j : integer;    -- Here is another
```

Block Comments

Block comments begin with “/*” and end with “*/”. The system ignores all characters between these two delimiters. Block comments can span any number of lines.

Example:
block comment

```
/*  Here is the start of a long multi-line comment that
  goes for several lines.  The next statement:
    x = 10;
  is not going to be executed, because it is in the comment  */
```

A block comment can contain a single-line comment or another block comment. This means that you can nest block comments. For example:

Example:
nested block comment

```
/*  Start first block comment
  /* Start second block comment
  End second block comment */
End first block comment */
```

Names

You use names to identify all the Forte system components, including:

- projects
- environments
- named constants
- classes
- interfaces
- service objects
- methods
- attributes
- events
- event handlers
- cursors
- variables
- parameters
- labels

A Forte name contains alphanumeric characters and underscores. The first character must be an alphabetic character or an underscore. Case is not significant. The name can be any length. When you name a new component in TOOL, the name must be unique for the current scope (see the next section for a description of scopes).

Restrictions

A Forte name can have no spaces or symbols except the underscore. You cannot use TOOL reserved words (see Appendix A, "Reserved Words") and you should avoid using SQL reserved words (also in Appendix A). Also, Forte's internal naming scheme imposes the following restrictions:

- You cannot start any name with "forte"
- You cannot end any name with "proxy"

In most TOOL statements, when you want to reference a component, you simply type its name. However, when you are using SQL statements, you may need to preface a Forte name with a colon in order to distinguish it from a column name. See "Using Forte Names with SQL" on page 53 for more information.

Use double quotes for names that are reserved words

If you need to reference an existing component whose name is the same as a reserved word, you must enclose the name in double quotes. (This might happen when you import a .pex file.)

Name Resolution

Unless you use a qualified name (described under "Qualified Names" on page 53), Forte does not consider the context in which you use a name. Instead, Forte searches for any component with that name. After finding the first component with the specified name, Forte then determines if it is the correct kind. If it is not, Forte generates an error.

Forte searches for the name starting in the current scope and moving out to the enclosing scopes. The order of scopes that Forte uses to resolve a name is shown in the following table.

Scope Search Order for Forte Name Resolution

Order	Component Checked	Named Items Checked
1	Current statement block	local variables, statement labels, named constants
2	Current method or current event handler	local variables, statement labels, named constants, parameters
3	Current class	attributes, methods, events, event handlers
4	Current project	classes, interfaces, project constants, cursors, service objects, supplier projects
5	Supplier plans	classes, interfaces, project constants, cursors, service objects A name from a supplier plan must be unique. If the same name refers to components in different supplier plans, you must use the plan name to qualify the name.

Because TOOL searches for a name starting at the current statement block and searching through the enclosing scopes, a name in an inner scope can "hide" a name in an outer scope. In the following example, the programmer declares a variable that has the same name as an existing class. When Forte executes the statements after the variable declaration, the class name is not recognized. Once outside the statement block that contains the variable declaration, the class name is again recognized.

Example:
name resolution

```
-- ... A class named Artist is defined ...
for ...
  artist : Artist = new;
  -- 'artist' is declared as a variable name for an object
  -- of Artist class.
  artist.Born = 1844;
  painter : Artist;
  -- ERROR: ARTIST IS NOT A CLASSNAME HERE
end for;
-- Now the name Artist is a class again.
painter : Artist;
```

Qualified Names

You can use dot notation to qualify a class element with a class name and/or plan name. Using fully qualified names allows you to explicitly identify a unique element. Fully qualified names are useful when the same name is used by multiple elements, and are helpful for code maintenance. Also, when you use a fully qualified name, TOOL searches in the specified project or library only, ignoring the default search order for the supplier plans.

Class name

To identify which class an element belongs to, you can use the following syntax:

class_name.*class_element_name*

Project name

Use the following syntax to identify the project or library to which a component belongs:

project_name.*project_component_name*

For class components in supplier plans, you can specify both the plan name and the class name, using the syntax:

plan_name.*class_name*.*class_element_name*

Using Forte Names with SQL

Use colons for Forte names

TOOL SQL statements reference database tables and columns as well as Forte names. Therefore, in almost all TOOL SQL statements you must prefix Forte variable names with colons to distinguish them from database names.

Example:
Forte names with SQL

```
year_born : integer = 1800;
sql select ptr_name into :name from painter_table
  where birth < :year_born;
```

In some cases you need not use colons before Forte names. The descriptions of the individual statements in Chapter 3, “TOOL Statement Reference,” indicate when colons are not needed.

If the Forte name is the same as a SQL reserved word

While you should avoid using Forte names that are the same as SQL reserved words, it may be necessary occasionally. If a Forte name is the same as a SQL reserved word, and you are not using a colon to identify it as a Forte name, then you must enclose it in double quotation marks. For a list of SQL reserved words, see Appendix A, “Reserved Words.”

Simple Data Types

You use the Forte simple data types, described in the next few sections, for handling string, boolean, and numeric data. You can use simple data types to declare variables, attributes, and parameters. The numeric data types are discussed together because they can be combined in numeric expressions.

Advantages of classes over simple data types

Forte provides an “object” version of each simple data type. The advantage of using an object to store data is that the class provides methods for manipulating the data. Forte also provides classes specifically for storing and manipulating dates and times, time spans, and images. See the Framework Library online Help for general information about using the class data types and reference information on the subclasses of the DataValue and DataFormat classes.

Forte provides additional numeric data types and data structures for methods that integrate with external systems, such as methods for C, DCE, or CORBA classes. The data structures are:

- pointer
- enum
- fixed-size array
- struct
- union

For information about the external data types and data structures, see *Integrating with External Systems*.

String Data Types

string data type

The string data type stores a character string. This data type is very simple. You use a string constant to specify a string value, but there are no string expressions. Although you can compare strings in boolean expressions (see “Boolean Data Type” on page 56), you cannot manipulate strings. It is usually preferable to use the TextData class, which provides many text handling methods and an unlimited text string.

To declare a data item with the string data type, use the **string** key word. For example:

```
s : string;
name : string = ‘Jones’;
```

The default value of a string data item is NIL, which means that it contains no string.

Also see the “Char Data Type” on page 55.

String Constants

A string constant is any series of characters enclosed by single quotes.

Example:
string constants

```
s : string;
s = ‘Jones’;
if s = ‘Smith’ then
  ... will not execute ...
end if;
```

To specify an empty string, use two single quotes with no characters between them. The following table specifies how to enter special characters within a string:

Special Character	How to Enter It
’	\’
\	\\
new line	\n
carriage return	\r
tab	\t
alert (bell)	\a
backspace	\b
formfeed	\f
vertical tab	\v
octal value	\000, where 0 is 0-7
hexadecimal value	\xhh, where h is 0-9, A-F, or a-f (the character with the hex value)

Char Data Type

The char data type exactly matches the C char data type and contains a single byte of data. TOOL provides this data type so that you can pass data to C procedures using the same data type that C requires for a character string—an array of char data items.

Because TOOL does not actually support a char constant, you can convert a string constant that contains one byte to type char. Remember that a string constant can also contain sequences, which can also be converted to a char data value if the string contains only one byte. The following example shows this conversion:

Example: converting a string constant to char data

```
myCharValue : char;
-- ‘a’ string constant in TOOL not char constant, as in C,
-- but TOOL converts the string constant to char data
myCharValue = ‘a’;
-- myCharValue = ‘ab’ invalid because ‘ab’ contains two bytes
myCharEscapeSequence : char;
-- ‘\n’ is new line escape sequence in single-byte constant
myCharEscapeSequence = ‘\n’;
```

TOOL also automatically converts char data to integer data when the char value is assigned to an integer type variable. The reverse is also true, as shown in the following example:

Example: converting char to integer

```
c : char;
i : integer;
-- 'a' is a char constant
c = 'a';
-- Convert char to integer value 97 (ASCII equivalent of ‘a’)
i = c;
-- Convert integer to char value ‘a’ (ASCII equivalent of 97)
c = i;
```

Unlike C, TOOL does not allow you to assign string constants larger than one byte (more than one character) to integer values.

If you are writing a multilingual application that must support multiple-byte character sets, remember that you cannot use char to contain a multiple-byte character. For more information about using char data in multilingual applications, see the *Forte 4GL Programming Guide*.

Boolean Data Type

The boolean data type contains two logical values, TRUE and FALSE. Use the boolean data type when a data item has only two values (such as true and false, yes and no, or on and off). To declare a data item with the boolean data type, use the **boolean** key word, as in the following example:

```
test : boolean;
test = FALSE;
test2 : boolean = TRUE;
```

The default value for a boolean data item is FALSE.

Boolean Constants

The boolean constants are the key words TRUE and FALSE. These can be in upper or lower case. The following example shows setting two variables to TRUE.

```
test : boolean;
test2 : boolean;
test = TRUE;   test2 = true;
```

Boolean Expressions

Boolean expressions are expressions that resolve to a logical value of TRUE or FALSE. You use boolean expressions to specify the conditions for several TOOL programming statements.

The following types of boolean expressions are described in the next sections:

Comparison expression Uses a comparison operator to compare two values (numeric, string, pointer, or object) and produce a value of TRUE or FALSE.

Logical expression Uses a logical operator to compare one or two boolean values and return a value of TRUE or FALSE.

In addition to using Boolean expressions, you can use the following classes to reference boolean values in TOOL code:

BooleanData and BooleanNullable Objects of the BooleanData class have a value of TRUE or FALSE. Objects of the BooleanNullable class have a value of TRUE, FALSE, or NULL.

Comparison Expressions

A comparison expression compares two numeric, two string, or two object values with a comparison operator to produce a value of TRUE or FALSE. The numeric or string values can be constants, variables, attributes, named constants, expressions, and methods that return an appropriate value. The following table describes the comparison operators.

Operator	Meaning	Description
=	Equals	Result is TRUE if left side is equal to right side. Defined for numeric data types, strings, pointers, and objects. Two pointer values are equal if they contain the same address. Two object values are equal if they reference the same object.
<>	Not equals	Result is TRUE if left side is not equal to the right side. Defined for numeric data types, strings, pointers, and objects.
<	Less than	Result is TRUE if left side is less than the right side. Defined for numeric data types and strings.
>	Greater than	Result is TRUE if left side is greater than right side. Defined for numeric data types and strings.
<=	Less than or equal to	Result is TRUE if left side is less than or equal to right side. Defined for numeric data types and strings.
>=	Greater than or equal to	Result is TRUE if left side is greater than or equal to right side. Defined for numeric data types and strings.

The following code fragment uses comparison expressions:

Example:
comparison expressions

```
x : integer = 10;
if x < 100 then
-- comparison expression in if
  ... this will be executed ...
end if;
-- Also, it can be a boolean value
test : boolean;
test = x < 100;
if test then
-- the same results as above
  ... this will be executed ...
end if;
```

Logical Expressions

A logical expression compares two boolean values with a logical operator to produce one boolean value, TRUE or FALSE. The boolean values you can use in the expression include comparison expressions, logical expressions, boolean constants, boolean variables, boolean attributes, and methods that return a boolean value. You can also compare two objects of the BooleanData or BooleanNullable class (see the Framework Library online Help for information on these classes). The following table describes the logical operators.

Operator	Description
not	Negates one boolean value. If the value is TRUE, **not** produces FALSE. If the value is FALSE, **not** produces TRUE.
and	Result is TRUE if both values are TRUE. If one or both values are FALSE, the result of the expression is FALSE.
or	Result is TRUE if either value is TRUE. The expression is FALSE only if both values are FALSE.

For BooleanNullable objects, a value of NULL is equivalent to FALSE.

The following code fragment uses logical expressions:

Example: logical expressions

```
if not x > 10 then
  ...
end if;
if (x > 10) or (x < 0) then
end if;
  ...
if (x > 10) and (y < 100) then
  ...
end if;
```

The “and” and “or” operators only evaluate both operands if necessary. In the following example, if obj is NIL, then obj.method1() will not be evaluated:

```
if obj <> NIL and obj.method1()
```

Operator precedence

The logical expression is evaluated with the following operator precedence (see “Numeric Expressions” on page 61):

Precedence	Operator
1	arithmetic and address operators
2	comparison operators
3	bitwise operators (see “Numeric Expressions” on page 61)
4	not
5	and
6	or

The following sample code shows operator precedence:

Example: operator precedence

```
x : integer = 1;
y : integer = 0;
if  x + y > y - x or not x > y then
  ...same as...
if  ((1+0) > (0-1)) or (not (1>0)) then
  ... or ...
if (1 > -1) or (not (TRUE)) then
  ... or ...
if (TRUE) or (FALSE) then
  ... or ...
if TRUE then
```

Parentheses in expressions

Use parentheses to guarantee the order of evaluation. TOOL evaluates the expressions in the innermost parentheses first.

Example: parentheses in an expression

```
if (((x > y) or (y < 2)) and (x > 2)) then
  ...
end if;
```

You can also use parentheses with the **not** operator. This negates the entire expression within the parentheses. The following example illustrates:

Example: parentheses with **not**

```
if  not (((x > y) or (y < 2)) and (x > 2)) then
  ...
end if;
```

BooleanData and BooleanNullable Classes

You can use a reference to a BooleanData or BooleanNullable object as a boolean expression in a TOOL statement. If the value of the object is TRUE, the expression is TRUE. Otherwise, the expression is FALSE. An object of the BooleanData class has a value of TRUE or FALSE. An object of the BooleanNullable class has a value of TRUE, FALSE, or NULL.

Using a reference to one of these objects is useful when you want to use the result of a method as the boolean expression in a control statement. A number of methods in the DataValue classes return BooleanNullable as a result.

Example:
use of BooleanData
and BooleanNullable

```
b : BooleanData = new(value = TRUE);
if b then
  ... will be executed ...
end if;
t : TextData = new (value = 'hello');
-- IsEqual returns a BooleanNullable
if t.IsEqual('hello') then
  ... will be executed ...
end if;
```

See “Objects” on page 64 for information about using an object reference in a TOOL statement. See the Framework Library online Help for information on the BooleanData and BooleanNullable classes.

Numeric Data Types

The numeric data types allow you to store integers and floating point numbers of different sizes. This section describes the integer and float data types, and provides general information on numeric constants and numeric expressions.

Integer Data Types

Of the integer data types, only some are guaranteed to be portable because they have the same range on every platform. The non-portable types use different representations on different machines. Keep this in mind when you declare integer data items. The following table lists the integer data types and indicates whether or not they are portable:

Key Word	Description	Portable
int	At least -32,768 to +32,767.	no
long	At least -2,147,483,648 to +2,147,483,647.	no
short	At least -32,768 to +32,767.	no
i2	Signed two byte integer. Exactly -32,768 to +32,767 all platforms.	yes
ui2	Unsigned two byte integer, 0 to +65,535.	yes
integer or i4	Signed four byte integer. Exactly -2,147,483,648 to +2,147,483,647 on all platforms.	yes
ui4	Unsigned four byte integer, 0 to +4,294,967,295.	yes
i1	Signed one byte integer, -128 to +127.	yes
ui1	Unsigned one byte integer, 0 to +255.	yes

To declare a data item of an integer data type, use the appropriate key word. For example:

```
i : integer;
j : short = 32;
```

The default value for an integer data item is 0.

uInt and uLong data types

Forte provides two additional data types specifically for methods that integrate with external systems, such as methods for C, DCE, or ObjectBroker classes: uInt for unsigned integer and uLong for unsigned long integer. For more information about these data types, see *Integrating with External Systems*.

Float Data Types

Forte supports two float data types; the exact precision of the float data type depends on your particular machine. The float data types are:

Key Word	Description
float	Approximately 10E-38 to 10E+38, with about 7 digits of decimal precision.
double	Approximately 10E-308 to 10+308, with about 15 digits of decimal precision.

If you want to ensure that your code is completely portable, you should only use the precision that is available on all the machines you plan to use. For precise decimal behavior, you can use the DecimalData class (see the Framework Library online Help for information).

To declare a data item with the float data type, use the appropriate key word.

Example: float types

```
i : float;
j : double = 10;
pi : double = 3.14159268;
```

The default value for a float data item is 0.0.

Numeric Constants

Integer constants

An integer constant is a sequence of digits between 0-9. No other characters are allowed. To indicate a negative number, use a minus sign. A number without a sign is considered positive but you can use a plus sign if you wish. The syntax is:

[+|-]*digits*

```
x : integer;
x = 10;
x = -23;
x = +43;
```

Hexadecimal and octal integers

You can use hexadecimal or octal constants to specify an integer. The syntax for hexadecimal integers is:

0x*hexdigit*

where *hexdigit* is:

0-9, A-F, or a-f (the character with the hex value)

The syntax for octal integers is:

0*octaldigit*

where *octaldigit* is any digit from 0-7. The first non-octal digit terminates the number.

```
x = 0x20;
x = 0xff04;
x: integer = 011; -- returns a value of 9
x: integer = 08; -- returns a value of 0
```

Float or double constant

A float constant is a sequence of digits 0-9 with a single decimal point (.). You can also include an exponent. To indicate a negative number, use a minus sign. A number without a sign is considered positive but you can use a plus sign if you wish. The syntax is:

[**+**|**-**]*digits***.***digits*[**e**|**E**[**+**|**-**] *integer*]

Example: numeric constants

```
y : double;
y = 10;
y = -123.456;
y = -1.3e+12;
```

Numeric Expressions

A numeric expression combines two numeric values with an arithmetic operator to produce one numeric value. The numeric values can be numeric constants, numeric variables, numeric attributes, methods that return a numeric value, SQL statements that return numeric values, and numeric expressions.

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
/	Division (integer results are truncated, not rounded)
- (unary)	Negative
+ (unary)	Positive
%	Mod
&	bitwise and
|	bitwise or
^	bitwise exclusive or
~	bitwise (unary)

Order of evaluation

Numeric expressions are evaluated from left to right, with the following operator precedence (from highest to lowest):

Precedence	Operator
1	[ ] ->
2	* (pointer dereference) & (address of)
3	- (unary) + (unary) ~
4	* / %
5	+ -
6	< > = >= <= < >
7	& (bitwise)
8	^ (bitwise xor)
9	| (bitwise or)
10	not
11	and
12	or

The following example illustrates the order of evaluation in a numeric expression:

Example: evaluation of numeric expression

```
x : integer = 1;
y : integer = 2;
z : integer;
z = x + y * y;   -- Evaluates to 5
```

Use parentheses to guarantee the order of evaluation. TOOL evaluates the expressions in the innermost parentheses first.

Example: parentheses in numeric expression

```
x : integer = 1;
y : integer = 2;
z : integer = (x + y) * y;
-- Evaluates to 6
```

Expression data type

The data type of a numeric expression's result is determined by the data type of both operands (left and right). The following table shows the data type for each possible pair of operands. Since the table is symmetric, the rows and columns can correspond to either the left or right operand:

	double	**float**	**ui4**	**long**	**integer/i4**	**int**	**i2**	**i1**	**ui2**	**ui4**
double	double	double	double	double	double	double	double	double	double	double
float	double	float	float	float	float	float	float	float	float	float
ui4	double	float	ui4	ui4	ui4	ui4	ui4	ui4	ui4	ui4
long	double	float	ui4	long	long	long	long	long	long	long
integer/i4	double	float	ui4	long	integer	integer	integer	integer	integer	integer
int	double	float	ui4	long	integer	integer	integer	integer	integer	integer
i2	double	float	ui4	long	integer	integer	integer	integer	integer	integer
i1	double	float	ui4	long	integer	integer	integer	integer	integer	integer
ui2	double	float	ui4	long	integer	integer	integer	integer	integer	integer
ui1	double	float	ui4	long	integer	integer	integer	integer	integer	integer

Forte automatically converts the value whose data type is different than the result type. This takes effect before the operation is executed.

Casting numeric types

Because Forte automatically performs conversions for the numeric values in expressions, you normally do not need to cast numeric types. However, for special cases, you can use the same syntax for casting numeric types as you do for casting objects. The syntax is:

numeric_type (*expression*);

or

(*numeric_type*) (*expression*);

For information on casting objects, see "Casting" on page 69.

Using SQL Statements in Numeric Expressions

You can use SQL statements that return an integer to specify an integer expression in any TOOL statement.The following SQL statements return integer values:

- sql delete
- sql insert
- sql fetch cursor
- sql select
- sql update
- sql execute immediate
- sql execute procedure

To use a SQL statement as a numeric expression, you must enclose the entire SQL statement within parentheses. The following example shows assigning the return value of a **sql fetch** statement to an integer expression:

Example: SQL statement in numeric expression

```
c : DefinedCursor;
sql open cursor c on session db_session;
while ((sql fetch cursor c into x, y, z) > 0) do
  ... another row was fetched ...
end while;
sql close cursor c;
```

Objects

By using a class name, you can define any variable, attribute, or parameter to have a data type of “object.” This means that the data item serves as a reference to an object. Both Figure 3 and the following line of code illustrate this.

```
a : Artist = new (name='Henri Rousseau');
```

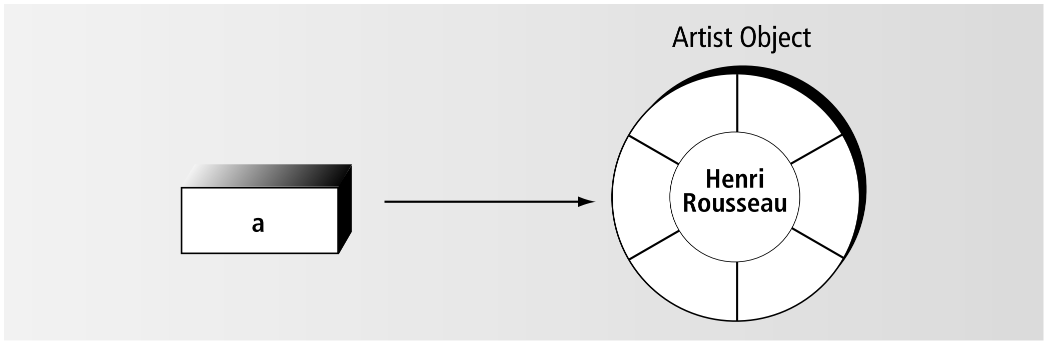


Figure 3 *Relationship between Data Item and Object*

More than one data item can point to the same object. The following example illustrates two data items pointing to the same object:

Example: two data items pointing to same object

```
t : TextData = new(value='abc');  -- TextData is a class
s : TextData;
s = t;  -- s now points to the same object as t
s.SetValue('def');  -- Change object value to 'def'
-- s and t are still pointing to the same object.
task.Part.LogMgr.PutLine(s);  -- will be 'def'
task.Part.LogMgr.PutLine(t);  -- will be 'def'
```

NIL

A data item can also have a value of NIL, which means “no object.” See “The NIL Constant” on page 69 for more information about NIL.

Working with Objects

In TOOL, you can manipulate an object as a whole or you can work with its individual attributes. For example, when you specify an object reference for a parameter, you are passing a reference to the entire object. When you set the value of an attribute, you are working with just one of the object’s attributes. The following table summarizes the syntax for working with an object and its components.

To:	Use this syntax:
Construct an object	{*variable* | *attribute*}= {**new** [(*attribute* = *value* [, *attribute* = *value*]...]) | *object_class* ([attribute = value [, attribute = value]...)]}
Reference an object	{*variable* | *attribute* | *method_invocation* | *array_row*[*n*] | *service_object*}
Specify an object value	{**NIL** | *object_constructor* | *object_reference*}
Reference an attribute	*object_reference***.***attribute*
Assign a value to an attribute	*object_reference***.***attribute* = *expression*
Cast an object	*class* (*object_reference*)

This section first describes how to work with entire objects in TOOL. It then describes how to work with two class components: attributes and methods.

Using Object Constructors

To create an object, you use an object constructor. Object constructors create new objects that you can then manipulate with TOOL code. Since the default value of an attribute, variable, or parameter with a class type is NIL, you must also create the objects they reference.

The object constructor syntax is:

{*variable* | *attribute* } = {**new** [(*attribute* = *value* [**,** *attribute* = *value*]...]) **|**
object_class ([*attribute* = *value* [**,** *attribute* = *value*]...)]}

new keyword

The simplest way to create an object is to use the **new** key word. This creates a new object, using default values for the attributes. The following example creates a new object, called painter, of the Artist class:

```
painter: Artist = new;
```

However, you cannot use the **new** keyword to create an object in the following cases:

- when creating a new object as a value for a parameter
- when creating a object nested in another constructor (see example below)

In these two cases, you must use the syntax that specifies the object class.

You can use an object constructor when you are assigning a value to a variable or an attribute of a class type.

Setting variable values

For variables, you can construct the object when you declare the variable or when you are assigning a value to the variable.

Setting attribute values

For attributes, you construct the object when you assign a value to an attribute. To specify values for attributes, you can enter a list, enclosed in parentheses, of attribute/value assignments. The value for an attribute can be any expression that is compatible with the data type of the attribute (including an object constructor). Forte sets the values in the order specified. Any attributes for which you do not specify a value are set to default values.

Example:
setting attribute values

```
painter : Artist;
painter = new(
  Name = TextData(value = ‘Henri Rousseau’),
  Country = TextData(value = ‘France’),
  Born = 1844, Died = 1910,
  School = TextData(value = ‘Primitive’),
  Comments = TextData(value =
    ‘He was ... pompous and absurd.’));
```

Shared and
transactional objects

Every class specifies whether objects of that class can be shared and/or transactional, and whether these settings can be overridden. If the settings can be overridden, you can set the object’s IsShared and IsTransactional attributes in the object constructor to override the default setting. (You can also set these attributes at any point after constructing the object.) See the Framework Library online Help for information about these attributes.

The following code sample illustrates setting the IsTransactional attribute:

Example: setting IsTransactional
attribute

```
t : TextData = new(value =
    ‘Hello’, IsTransactional = TRUE);
x : TextData = new(value = ‘Goodbye’);
  ...
-- Set after construction
x.IsTransactional = TRUE;
```

Declared type and runtime type

When you declare a variable, attribute, or parameter of a class type, the class for the data item is its *declared type*. However, the object that the data item points to can either be of the same class as or a subclass of the declared type. For example, you could declare a variable of class FieldWidget, and then assign it to reference an object which is actually of type TextField, a subclass of class FieldWidget (assigning object values is described under "Specifying an Object Value" on page 68). The class of the actual object is called the *runtime type*.

The optional "object class" in the object constructor allows you to declare an object with a different runtime type than the declared type of the variable. This can be any subclass of the variable's class. The following example illustrates:

```
n : NumericData;
n = DoubleData(value = 123.45);
```

The advantage of making the declared type of a data item a higher level than its runtime type is that your code is reusable. A single data item can point to many different classes of objects. See "Casting" on page 69 for further information about declared and runtime types.

Setting parameter values

If you want to use an object constructor as an expression, for example, to set the value of a parameter, you must include the object class. You must include the class because in this context Forte needs to know the object's type.

Example: object constructor as expression

```
-- ... from the AuctionMgr OpenNewList method ...
-- The AddBid method takes Painting and DecimalData
--  parameters.  self refers to the current object.
self.AddBid (paintingToAdd = t_painting,
  startingBid = DecimalData(value = 2000000.00));
```

The Init Method

Whenever you construct a new object, Forte performs the following (in order):

1. Creates an empty object of the given class, using zero for numeric values, empty for string values, and NIL for object values.
2. Invokes the Init method on the object.

   Every class inherits the Init method from the Object class. You can override this method for your own class. The Init method you write will then automatically be invoked every time you construct an object of that class. If you do override the Init method, be sure to invoke super.Init as the first statement of the Init method body.
3. Sets the values of the attributes that you have specified in the attribute/value list. Forte performs these assignments in the order in which you specify them.

The following code sample illustrates an Init method:

Example: Init method

```
-- The Artist Init method
super.Init();
-- Always execute Super's Init.
-- Create empty TextData objects, as Transactional too.
self.Name = new(IsTransactional = TRUE);
self.School = new(IsTransactional = TRUE);
self.Country = new(IsTransactional = TRUE);
self.Comments = new(IsTransactional = TRUE);
self.Born = 0;   self.Died = 0;
```

Referencing an Object

The term “object reference” can refer to any one of several means of identifying an object.

To reference an object as a whole, you use the name of a data item that points to the object. In the following example, “t” and “s” are two variables that point to the same TextData object:

Example: referencing an object

```
t : TextData = new(value = ‘Hello’);
s : TextData = NIL;    -- References no object.
s = t;  -- Now both ‘s’ and ‘t’ reference same object.
```

TOOL provides five special key words for referencing special objects:

Key Word	Description
application	Refers to the current application (used to reference the message catalog for the current application—see *Forte 4GL Programming Guide* for information).
self	Refers to the current object, that is, the object on which the current method was invoked.
super	This key word is only used in the format “**super**.*methodname*” to refer to an inherited method (more specifically, a method for which you have written an overriding method). See “Invoking Methods” on page 73.
task	Refers to the current TaskHandle object for the current task.
transaction	Refers to the current TransactionHandle object for the current transaction.

The following example illustrates using the **transaction** key word:

```
-- ... Abort the current transaction for the task ...
transaction.Abort(true);
```

You can use an expression to identify the object. For example, if a method has an object as its return value, you can invoke the method to provide an object reference. See “Invoking Methods” on page 73 for more information.

Dot notation

To reference an object that is an attribute of another object, you use dot notation. The syntax is:

*object_reference***.***object_reference*[.*object reference*...]

For example:

```
-- ... in a UserWindow class, change the title...
self.Window.Title = new(value = ‘This is a title’);
```

If an object reference is long and you need to continue a command on a new line, you can use a return character before or after any period in the object reference. For example:

```
--... in a UserWindow class, change the title...
self
    .Window.
    Title = new(value = ‘This is a title’);
```

Strings are pointers to objects

If you assign a TextData object to a string, the string becomes a pointer to the value attribute of the TextData object. If the value of the TextData object subsequently changes, the value of the string also changes accordingly, as the string is merely a pointer to the TextData object's value attribute.

Array row reference

To reference an object that is an attribute of an array, you must use a special notation to reference the array row. See “Array Classes and Array Objects” on page 77 for information about referencing the individual objects in an array.

Example: referencing an object in an array

```
a : Array of Bid;
if a[1].BidInProgress
-- TRUE if bidding active on this one
  ...
end if;
```

Objects for Widgets

A Guide to the Forte 4GL Workshops describes how certain widgets are associated with two attributes: the attribute that contains the data for the widget and the attribute that points to the widget object. The widget object contains information about the behavior and appearance of the widget.

Because both of these attributes share the same name, it is impossible to reference both attributes by simply using the attribute’s name. In order to distinguish between the two related attributes, you refer to the attribute containing the widget data using the attribute name by itself. You refer to the attribute containing the widget object using the attribute name in angle brackets as follows:

<*widget_attribute*>

The following code sample references both the widget object itself (<BidStatusMessage>) and the widgetdata (BidStatusMessage):

Example: widget reference

```
-- ... from the Display method for ViewPaintingWindow...
-- The message field
<BidStatusMessage>.State = FS_INVISIBLE;
-- The message data
BidStatusMessage.SetValue(‘Bid in Progress.’);
```

Child fields

If a widget is a child of a named compound widget that is mapped to a class, you need to use the parent’s name in order to identify it. Use dot notation as follows:

<*parent_widget_attribute.child_widget_attribute*>

Specifying an Object Value

To specify a value for a variable, parameter, or attribute of a class or interface type, you must use a reference to an existing object, an object constructor, or the special NIL constant. For example:

Example: object values

```
t : TextData = new(value = ‘Hello’);
s : TextData = NIL;
s = t;
```

Object value for class type

If the data item has a class type, the object value must be the same class or a subclass of the data item’s declared type. In other words, the class for the object value on the right of the assignment must be the same class as or a subclass of the one on the left.

Object value for interface type

If the data item has an interface type, the object value must be of a class that implements the interface (or a subclass of a class that implements the interface). The implementing class is the data item’s runtime type.

Object constructors

When you are specifying an object value for a variable, parameter, or attribute, you can use an object constructor (described under “Using Object Constructors” on page 65) to create a new object. The following example illustrates:

```
t : TextData;
t = new(value = ‘Hello’, IsTransactional = TRUE);
```

Comparing Objects

Simple data types

If the individual variables a and b are assigned to simple data types, each assignment is an assignment to a value. And, a test of equality or inequality (a = b or a <> b) is a comparison of values when a and b are simple types.

Class data types

In contrast, if the variables c and d are assigned to class types, each assignment is a reference to an object. A test of equality or inequality is a comparison of references to objects.

Thus, when dealing with objects, only if two variables refer to the *same* object will a comparison of equality return TRUE. If two variables refer to two objects that are identical (same class and same values), but not the same object, a comparison of equality returns FALSE. In this case, the two objects are “identical” but they are not equal.

The NIL Constant

The NIL constant is a special object value that means “no object.” You can assign the NIL value to a data item of any class. When no data item is left that points to an object, Forte automatically frees the memory used by the object; there is no explicit “free” operation. When a data item has a value of NIL, you cannot perform any operations on it. If you try to invoke a method on the object, set one of its attributes, or use it in an expression, you will get an exception.

Example: NIL data item

```
t : TextData = NIL;
if t.IsEqual(‘Hello’) then
-- ERROR: WILL GET EXCEPTION!!!
```

Using Methods as Object Values

When a method returns an object, you can invoke the method to specify an object value.

Example: using method as object value

```
t : TextData = new(value = ‘a’);
t.Concat(‘b’).Concat(‘c’).Concat(‘d’);
-- ‘t’ contains ‘abcd’
```

If you are assigning the return value of the method to a data item, it must be the same class or a subclass of the data item’s class.

(See “Invoking Methods” on page 73 for information about return values for methods.)

Casting

As described under “Specifying an Object Value” on page 68, when you assign an object value to a data item, the object can be a subclass of the data item’s declared type. For example, when you use a class parameter for a method, the object value you assign to the parameter can be a subclass of the parameter’s class. This allows you to use the parameter to pass objects that belong to any of its subclasses.

For example, the TextData class has a ReplaceParameters method with a parameter1 parameter with a type of DataValue. When you invoke the ReplaceParameters method, you can use the parameter1 parameter to pass an object of the DataValue class or any of its subclasses, including TextData or IntegerData, as follows:

Example: parameter value that is a subclass

```
t : TextData = new;
name : TextData = new(value = ‘mona’);
t.ReplaceParameters
  (source = ‘AM.RS: <%1> Return status is <%2>.’,
  parameter1 = name,
  parameter2 = IntegerData(value = 2));
```

The advantage of making the declared type of a data item a higher level than its runtime type is that your code is reusable. A single data item can point to many different classes of objects.

Other cases when the object value might be a subclass of the data item’s declared type are:

- the object value for an event parameter
- the object value for a method return value

The result of this assignment is that the declared type of the data item is different than its runtime type (the class of the object that the data item is pointing to). In the previous example, assigning the name object value to the parameter1 parameter means that the declared type of the parameter is DataValue while its runtime type is TextData.

When you reference a data item, the compiler recognizes only the declared type. Therefore, you will get an error if you try to set attributes, invoke methods, or post events defined only for the object’s runtime class. In order to manipulate the object, you must specify the actual class of the object.

To access attributes, methods or events defined only for the runtime class, you must cast the object. Casting simply means identifying the class of a particular object. First you specify the class name and then you specify the specific object you wish to cast. The syntax is:

class_name (*object_reference*) or (*class_name*) (*object_reference*)

In the following example we cast object dv from a Datavalue to either a DateTimeData or an IntegerData, based on the results of the IsA method:

Example: casting

```
-- ... from the DisplayData method of the Isql class ...
td : TextData;
dv : DataValue = row.GetValue (position = colnum);
length = self.CalcDataLength (data = dv, column = c);
if dv.IsA(DateTimeData) then
  -- For DateTimeData, use FormatDate, using Cast.
  td = self.dateFmt.FormatDate(DateTimeData(dv));
elseif dv.IsA(IntegerData) then
  -- For IntegerData, use FormatNumeric, using Cast.
  td = self.intFmt.FormatNumeric(IntegerData(dv));
...
end if;
```

When you cast an object, the compiler checks to see whether the cast has any chance of succeeding at runtime. It does this by checking that the casting type is actually the same class or a subclass of the data item's declared type. For example, the following is illegal and produces a compile-time error because TextData is not a subclass of IntegerData:

Example:
casting compile-time error

```
i : IntegerData
-- Illegal casting
...TextData(i)...;
```

You must be sure that the object you are casting really belongs to the class to which you are casting it. If it does not, the program will get a runtime exception.

Casting does not change the declared type of the data item. You must cast the object each time you reference it. Therefore, it is a good idea to cast the object once and assign it to a variable whose declared class is the same as the object's class, as in the following example:

Example:
casting and assigning

```
-- ... get the data from the clipboard, only if text ...
o : object;
o = self.Window.WindowSystem.CopyFromClipboard();
if o.Isa(TextData) then
  -- Must be text.
  t : TextData = TextData(o);
  --... now manipulate text from clipboard as ‘t’.
end if;
```

Casting should be needed infrequently. If you find that you are casting often, you should check your class hierarchy. You can move some of your methods to a higher level in the hierarchy. Or you can override the methods so casting will be unnecessary (when you use overriding, Forte uses the runtime type of the object to determine which method to use).

For specific information on casting arrays, see "Array Classes and Array Objects" on page 77.

Interfaces and casting

Because the interface is the declared type of a data item, the only operations available for the object are those defined by the interface. To access operations that are defined for the class but are not defined for the interface, you must first cast the object to its runtime type. Casting an object from an interface to a class is exactly the same as casting an object from a superclass to a subclass.

Accessing Attributes

To reference an individual attribute of an object, use the following dot notation:

object_reference.attribute

The object reference identifies the object. The attribute is any attribute defined for the object's class or inherited from its superclasses.

Example:
referencing attributes

```
painter : Artist = new;
painter.Born = 1843;
painter.Name = TextData(‘Vincent Van Gogh’);
```

Virtual attributes

You reference a virtual attribute using the same syntax. You do not need to be concerned about how the attribute is implemented.

No object reference
is the same as self

Note that if the attribute you are referencing is in the current object, you do not need to include the object reference. (By "current object," we mean the object on which the current method is operating.) When you specify an attribute name by itself, TOOL assumes the current object.

The following two examples have the same effect. The first example uses the **self** variable to reference the current object. The second example uses an attribute name without an object reference to reference the current object.

Example: current object

```
-- ... in the Init method for Artist ...
super.Init();
self.Name = TextData(IsTransactional = TRUE);
self.Born = 0;
 ...
-- ... is the same as ...
super.Init();
Name = TextData(IsTransactional = TRUE);
Born = 0;
...
```

Object and array attributes

The individual attributes of an object can themselves be objects or arrays. If the attribute you are referencing is an object, you can use the same dot notation to access any of its attributes.

Example:
object and array attributes

```
  -- ... In the Bid class ...
  b : Bid = new;
  b.PaintingForBid = new;
  b.PaintingForBid.Born = 1844;
  b.PaintingForBid.Name = TextData('Henri Rousseau');
```

If the attribute you are referencing is an array object, you need to use a special notation to reference an individual row in the array. See “Working with Array Rows” on page 82 for information.

Setting Attributes

To set the value of an attribute, use an assignment statement to assign a new value to the attribute.The syntax is:

object_reference.*attribute* = *value*

The object reference identifies the object. The attribute is any attribute defined for the object’s class. The value is any expression that has a data type compatible with that of the attribute. If the attribute is a class type, the value can be a reference to an existing object or an object constructor.

Virtual attributes

You set virtual attribute’s value using the same syntax. You do not need to be concerned about how the attribute is implemented.

In many cases, setting an attribute means more than just updating some data. The new value you specify may actually change the appearance or behavior of the object. For example, changing the FillColor attribute of field causes it to change color on the user’s screen. Sometimes this may even cause side effects for other objects.

Setting Attributes for Widgets

As described under “Objects for Widgets” on page 68, you use angle brackets to reference a widget object. Therefore, to set the value of a widget object’s attribute, you must use angle brackets in the assignment statement. The syntax is:

<*widget_attribute*>.*attribute* = *value*

Invoking Methods

To invoke a method, use an object reference to identify the object and then use dot notation to specify the method name. You must also specify a value for any required input parameters. If the method has a return value, you can use the method to specify a value in any TOOL statement.

The syntax is:

*object_reference***.***method*[(*parameter_list*)]

Object reference

The object reference identifies the object on which you wish to invoke the method. The method will perform its operations on this object.

Example: invoking a method

```
-- ... invoke Concat on a TextData object ...
t : TextData = new;
t.Concat('xyz');
```

To invoke a method on a widget object, use the angle bracket syntax to identify the object. The following example illustrates:

```
-- Set the selection area in the <Comments> TextField ...
<Comments>.SetSelectionRange(1,30);
```

No object reference is the same as self

Note that if the method is for the current object, you do not need to include the object reference. (By "current object" we mean the object on which the current method is operating.) If you specify a method name by itself, TOOL assumes the current object.

Method name

The method name identifies the particular method to invoke. This can be any method defined or inherited by the object's class. If there is more than one method with the same name, Forte checks for method overloading at compile time. At runtime, Forte checks for method overriding.

Overloaded methods

At compile time, if there is more than one method with the same name for the declared class of the object, Forte uses the parameter data types to determine which method to use (see "Parameters" on page 74 for information on method parameters). If there is no exact match between the data types of the parameter you specify and the declared parameters for the method, Forte uses the method with the best match. If there is no best match, you will receive a compile time error message.

Overridden methods

At runtime, Forte checks the runtime type of the object to see if the method has been overridden. If the declared class and the runtime class both have methods with the same name and parameter data types, Forte uses the method defined in the object's runtime class.

Use **super** to invoke an overridden method

When you override a method, you replace an inherited method with a new method with the same name. If you need to invoke the inherited method, simply use the **super** key word before the method name to specify the overridden method rather than the overriding method:

Example: use of **super**

```
-- ... in the Init method for the Artist class ...
super.Init();
self.Name = new;
...
```

This is particularly useful when you are writing a new method that adds functionality to an inherited method.

Parameters

When you create a method in the Class Workshop, you define each method parameter in the Method Properties dialog. You define each parameter as for input only, output only, or for both input/output. You also define a type for each parameter, specifying either a simple data type or a class.

When you invoke a method, the values that you specify for the method's parameters must correspond one to one with the parameters in the method definition. For input parameters, you can specify any value that is compatible with the parameter's data type. For output or input-output parameters, you can specify any attribute or variable that is compatible with the parameter's data type.

Example:
method parameters

```
-- ... The CopyRange method defined on TextData ...
t : TextData = new(value = 'abcdefg');
s : TextData;
s = t.CopyRange(startOffset = 0, endOffset = 3);
-- Will be 'abc'
```

Strings and
TextData parameters

If the method was defined with a TextData parameter, you can specify a simple string as the value when you invoke the method. Forte automatically creates the TextData object, using the string for its TextValue attribute.

Methods have required parameters, optional parameters, or a combination of optional and required parameters.

Required parameters

A parameter is required if the definition for the parameter does not specify an initial value. If the method has required parameters, you must specify values for all the required parameters when you invoke the method.

To indicate how the values and parameters correspond, you can use the parameter names, the parameter positions, or both. The syntax is:

[*name* =] *value* {, [*name* =] *value*}

To use the parameter names, enter the parameter name followed by the value. These can be in any order. We recommend using parameter names rather than positions because it makes your code easier to maintain.

Example:
using parameter names

```
-- ... The CopyRange method defined on TextData ...
t : TextData = new(value = 'abcdefg');
s : TextData;
s = t.CopyRange(startOffset = 0, endOffset = 3);
-- Will be 'abc'
-- ... is exactly the same as ...
s = t.CopyRange(endOffset = 3, startOffset = 0);
-- Will be 'abc'
```

To specify the parameters by position, enter the values in the same order as the corresponding parameters in the method definition. This order must be identical to the parameters in the original definition.

Example:
using parameter positions

```
-- ... The CopyRange method defined on TextData ...
t : TextData = new(value = ‘abcdefg’);
s : TextData;
s = t.CopyRange(0,  3);
-- Will be ‘abc’
```

To use a combination of the two techniques, you must first enter the values by position. You can then use parameter names for the remaining values in any order.

An input parameter is optional if the definition for the parameter specifies an initial value. If the method has optional input parameters, you can use or ignore any of the optional parameters. If you decide to ignore all the input parameters and there are no required parameters, you can omit the parentheses when you invoke the method.

If you are using parameter names, you can leave out any of the optional parameters. If you are using parameter positions, you can only leave out optional parameters that are at the end of the series.

Example:
optional parameters

```
-- ... The CopyRange method defined on TextData ...
t : TextData = new(value = ‘abcdefg’);
s : TextData;
s = t.CopyRange(startOffset = 3);
-- Will be ‘defg’
-- ... is exactly the same as ...
s = t.CopyRange(3);
-- Will be ‘defg’
```

Output Parameters

The value for any output parameter (either input-output or output only) must either be a variable or an attribute. Because the method that you are invoking will be assigning the output value to the output parameter, the input value you specify must be a data item that can be on the left side of an assignment statement.

Note that Forte passes output parameters by value result. This means that the value of the parameter is not changed until *after* the method returns.

Class Parameters

If a parameter whose type is a class is for input only, you can pass a reference to an object of the same class or a subclass of the parameter's declared class. (Note that when the declared class of the parameter is different than its runtime class, you may need to cast it before you reference it within the method. See "Casting" on page 69).

However, if a parameter whose type is a class is for input-output or for output only, you must pass a reference to an object of the same class as the parameter's declared class. Anything else is illegal.

Note Because you are passing a *reference* to an object, not the object itself, even if a parameter is for input only, if the method makes changes to the object, these changes are reflected when you return from the method. This is because both the invoking method and the invoked method are referencing the same object.

If the parameter was defined using the **copy** option, TOOL passes a reference to a copy of the object (see *A Guide to the Forte 4GL Workshops* for information about using the **copy** option when defining method parameters.)

Return Value

When a method produces a return value, you can invoke the method to specify a value in any TOOL statement. The only restriction is that the return value's data type must meet the requirements of the expression. (The return type is specified in the original method definition.)

In the following example, the programmer assigns the method's return value to a variable. This allows the programmer to reference the value through the rest of the current method.

Example:
using return value

```
-- In TextData, copy part of the string to another object
t : TextData = new(value = ‘abcdefg’);
s : TextData;
-- Note that the CopyRange method on the TextData class
-- is defined to return a TextData object:
-- CopyRange(...parms...) : TextData;
s = t.CopyRange(0,3);
-- Will be ‘abc’
s.Concat(‘cba’);
-- Now will be ‘abccba’
```

This variable can be any previously declared local variable with an appropriate data type for the return value.

When the return value is an object, the object's runtime class may be a subclass of the declared return type. When this is true, you must cast the return value before you can manipulate it in the current method (see "Casting" on page 69).

Array Classes and Array Objects

This section provides conceptual background on the Forte array classes, and provides information about working with array objects in your TOOL code. For information about C-style arrays, see *Integrating with External Systems*.

A Forte array class is a special class for storing and manipulating a collection of objects. The objects in an array are either of the same class or they share the same superclass. The data for an array is stored in the collection of objects. Figure 4 illustrates a collection of objects of the Artist class.

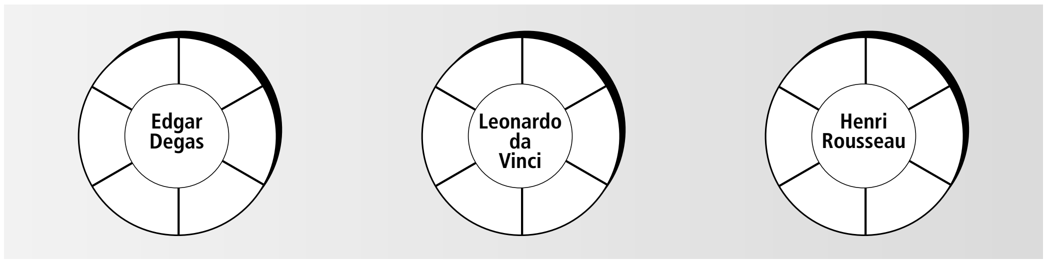
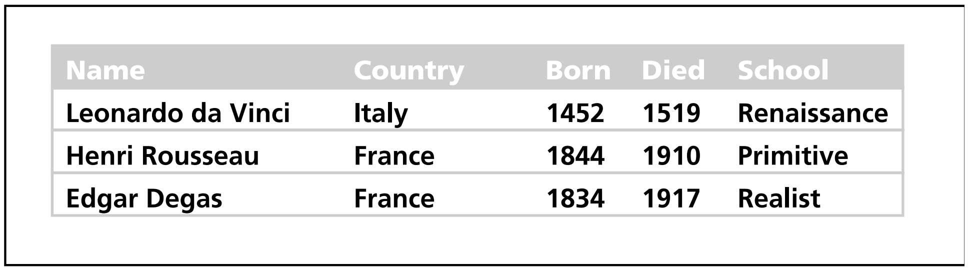


Figure 4 *Array Data*

Array data

You can think of an array as a table of data. Each object is like a row in the array or table, and each attribute is like a column. Figure 5 illustrates the array data:



Name	Country	Born	Died	School
Leonardo da Vinci	Italy	1452	1519	Renaissance
Henri Rousseau	France	1844	1910	Primitive
Edgar Degas	France	1834	1917	Realist

Figure 5 *Array Data as a Table*

Array object

Because an array is a single object that contains pointers to all of the array data, you can use the array to reference the set of objects as a unit. Or you can manipulate the individual objects. The array provides methods for manipulating the set of objects, such as methods for inserting rows and deleting rows. Figure 6 illustrates the array object and the array data:

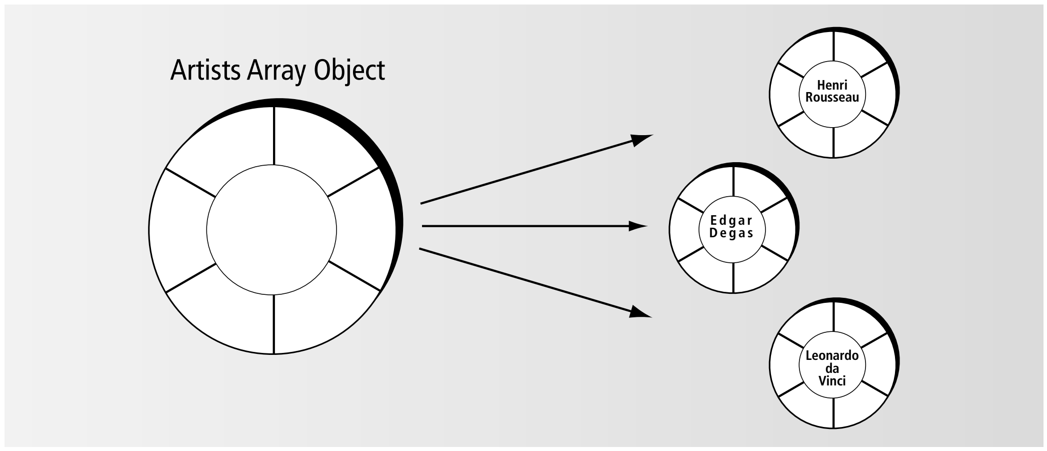


Figure 6 *Array Object and Array Data*

When you declare a variable, attribute, or parameter as an array, you are defining a pointer to the array object.

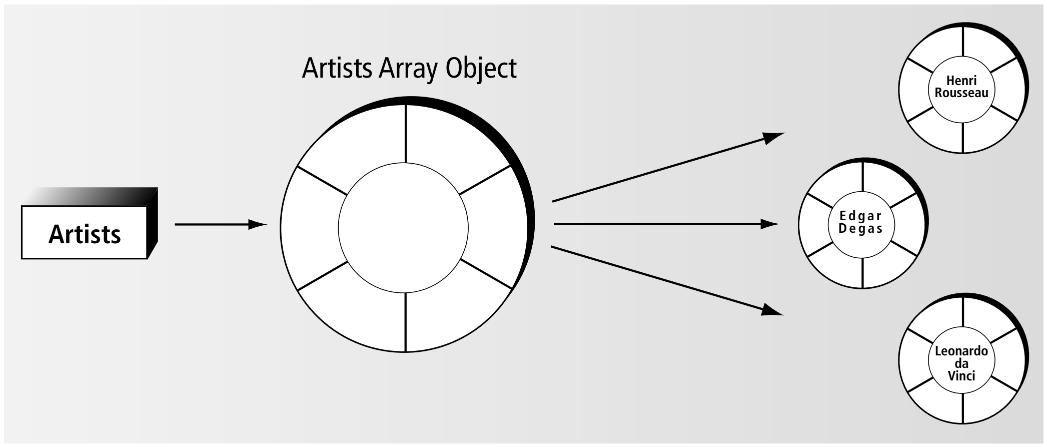


Figure 7 *Array Variable, Array Object, and Array Data*

You will use three Forte array classes: Array, LargeArray, and GenericArray. This section provides information about using TOOL to construct array objects, assign values to arrays, and reference array components. For information on the attributes and methods available for the classes, see the Framework Library online Help.

Working with Arrays

In TOOL, you can work with an array as a whole or you can work with its individual components. For example when you assign a value to an array parameter, you are passing a reference to the entire array. When you assign the value of one of the objects in the array, you are working with one of the array's individual components.

This section first describes how to work with entire arrays in TOOL. It then describes how to work with array components, rows, and attributes.

Declaring an Array Variable

Array and LargeArray classes

You will use three different kinds of arrays: Array, LargeArray, and GenericArray. The Array class is most efficient for smaller arrays, while LargeArray class is optimized for larger arrays (a rule of thumb is to consider over 255 rows as “large”). GenericArray is the superclass for Array and LargeArray, and is useful when you do not know what the runtime type of the array will be. Use whichever class is appropriate for the particular variable you are declaring. The syntax for declaring an array variable is:

name : {**Array** | **LargeArray** | **GenericArray**} **of** *class*;

Class for array

The class for the array specifies the class for the objects (or “rows”) that will be stored in the array. This can be any Forte class or custom class.

Declared type for array

Like any object, an array has both a declared type and a runtime type. The declared type of the array is a type made up of two classes, the array type (Array, LargeArray, or GenericArray) and the row type (any class).

See “Variables” on page 85 for information about declaring variables.

Constructing the Array Object

Before you can use an array you must construct the array object. You can construct the array object when you declare an array variable, or when you assign a value to an array variable, attribute, or parameter. If you do not use the array object constructor when you declare the data item, the value of the data item is NIL (that is, no array is initialized).

The syntax for the array object constructor is:

= **new;**

or

(*array_class* **of** *row_class*) ();

The following example shows two examples of using an array object constructor in a variable declaration:

```
  paintingsForBid = Array of Painting = new;
  paintingsForBid = LargeArray of Painting = new;
```

Adding array rows

After you construct the array object, you must add the rows to the array. You can do this either by using the AppendRow or InsertRow methods, or by assigning an object value to a nonexistent row. The following example illustrates adding an array row through assigning an object constructor to a nonexistent row:

Example:
adding array rows

```
-- Create and populate an array of TextData objects
txt_array : Array of TextData = new;
txt_array[1] = TextData(value = ‘first row’);
txt_array[2] = new(value = ‘second row’);
-- ‘new’ same as ‘TextData’
```

See “Working with Array Rows” on page 82 for more information about adding rows to the array by assigning an object to a nonexistent row. See the Array and LargeArray classes in the Framework Library online Help for information on the AppendRow and InsertRow methods.

Referencing an Array Object

To reference an array object as a whole, you assign an attribute or variable name to point to the array object.

Example:
referencing an array object

```
-- ... Test to see if array is empty ...
txt_array : Array of TextData = new;
txt_array[1] = new(value = ‘first row’);
if txt_array.Items > 0 then
-- Will be 1 here
  ... process ...
end if;
```

You can also use an expression to identify the array. For example, if a method has an array as its return value, you can invoke the method as an object reference.

If the array object that you want to reference is an attribute of another object, use dot notation to reference the array.

Example:
using dot notation

```
a_mgr : AuctionMgr = new;
a_mgr.PaintingsUnderBid = new;
-- Create empty bids
a_mgr.PaintingsUnderBid[1] = new;
-- Enter a new bid in array
if a_mgr.PaintingsUnderBid.Items > 0 then
-- Will be 1
  ... process ...
end if;
```

If the array that you want to reference is an attribute of another array, you must specify the row number for individual attribute (see "Referencing an Attribute in a Row" on page 84).

Specifying Array Object Values

To specify a value for an array variable, attribute, or parameter, you can use an array object reference, an array object constructor, or the NIL constant.

Example:
specifying array values

```
txt_array : Array of TextData = NIL;
if txt_array.Items > 0 then
-- ERROR.  EXCEPTION AT RUNTIME!

txt_array = new;
... is the same as ...
txt_array = (Array of TextData) ();

other_array : Array of TextData = NIL;
other_array = txt_array;
-- Both refer to same array
```

Runtime type for array object

The array object must be the same class or a subclass of the declared type of the array variable, attribute, or parameter. That is, the array type (Array, LargeArray, or GenericArray) must be the same class or a subclass of the declared array type, and the row type must be the same class or a subclass of the declared row type.

Example: declared
and runtime array type

```
v1 : Array of TextData;
v2 : GenericArray of DataValue;
v3 : LargeArray of IntegerData;
v4 : LargeArray of TextData;
-- m1 is a method that returns an Array of TextData

v2 = v1;
-- legal because declared type of v1 is Array of TextData
-- and declared type of v2 is GenericArray of DataValue
-- Array is a subclass of GenericArray and IntegerData
-- is a subclass of DataValue

v2 = v3;
-- legal because LargeArray is a subclass of GenericArray
-- and IntegerData is a subclass of DataValue

v2 = someObj.m1();
-- The declared type of m1 is the same as the declared
-- type of v2

v1 = v3;
-- ILLEGAL because LargeArray is not a subclass of Array

v4 = v3;
-- ILLEGAL because IntegerData is not a subclass of
-- TextData
```

Note that an array value on the right side of an assignment statement can function as an object as well because the array is an object. The following example illustrates this, using the same variables declared in the previous example.

Example: using superclass of GenericArray

```
o1 : Object;
o2 : Array;
o3 : GenericArray;
o4 : LargeArray;

o1 = v1;
-- legal because the declared type of v1 is treated
-- as Array, and Array is a subclass of Object
```

If the runtime type is not an array type, you can cast it to an appropriate type. (See "Casting Array Objects" on page 81 for information on this.)

Array object constructor

When you are specifying an array value for a variable, parameter, or attribute, you can use an array object constructor (described under "Constructing the Array Object" on page 78) to create a new array.

NIL

The NIL constant is a special object value that is equal to "no object." You can assign this value to a data item to remove the pointer to the current array object. When an array data item has a value of NIL, you cannot perform any operations on it. If you try to invoke a method on the array object or use it in an expression, you will get an error.

Casting Array Objects

As described earlier in this chapter, the declared type of an array object is a type made up of two classes, the array type (Array, LargeArray, or GenericArray) and the row type (any class). Forte allows you the cast the row type, the array type, or both. The syntax is:

(*array_type* **of** *subclass*) (*data_item*)

array_type is:

Array
LargeArray
GenericArray

Casting the row type

Casting an array row is useful when you have defined a parameter or variable that is an array of a general class to maximize reusability, and in some specific case you know that the array is actually made up only of elements of a known subclass.

DefaultClass attribute

The DefaultClass attribute of the array object contains the row type of the elements of the array. You can use this DefaultClass attribute to determine the runtime type of the array elements. The following example illustrates this.

Example: casting the row type

```
at : array of TextData;
  ... get data for ‘at’ ...
ad: array of DataValue;
ad = at;
if ad.DefaultClass = TextData then
  -- Will be processed. Must cast ad to array of TextData
  atmp : Array of TextData = (Array of TextData) (ad);
  atmp[1].Concat(‘more’)
end if;
```

When Forte casts the row type, it checks the DefaultClass attribute to verify that the casting type is the same class as or a subclass of the DefaultClass type.

If the DefaultClass attribute is not set because the declared type does not specify the row type (for example, l : LargeArray), the compiler automatically passes the cast. Therefore, you must be careful to cast the rows to their actual runtime type.

You must be sure that the rows you are casting really belong to the class to which you are casting them. If they do not, the program will get a runtime exception.

Casting the array type

Although it is not often necessary, you can cast the array type to any appropriate array type (one that is a subclass of the declared type). Array type casting might be useful if you are working with a generic object that you know is actually an array. The following example shows how to cast the array type:

```
o: Object;
...(GenericArray of TextData) (o)...
```

Casting both row and array type

If both the row type and array type need casting, you can use the same syntax to cast both at the same time. Forte first performs the cast on the array type. If this passes, it performs the cast on the row type.

```
sample : GenericArray of DataValue;
...(LargeArray of TextData) (sample)...
```

Working with Array Rows

Row numbers

The objects in the array can be considered numbered rows, starting with 1. The last row has the same number as the number of items in the array (the Items attribute of the array object contains the total number of rows in the array).

Referencing a row

To reference a single object in the array, specify the row number in square brackets.

array_reference[*n*]

The following code sample illustrates referencing arrays:

Example: referencing array rows

```
painters : Array of Artist = new;
painters[1] = new;
painters[2] = new(born = 1844, died = 1901,
  name = TextData(value = ‘Vincent Van Gogh’));
one_painter : Artist;
one_painter = painters[2];
```

An array row reference is an object reference

Because each array row is an object, you can use a reference to an array row anywhere you can use an object reference. For example, you could invoke a method on an array row or assign a new object to it.

Use this same syntax if the individual row you want to reference is an attribute of an object or an attribute of an array row.

Example: referencing an array that is an attribute

```
a_mgr : AuctionMgr = new;
a_mgr.PaintingsUnderBid = new;
a_mgr.PaintingsUnderBid[1] = new;
-- This is a Bid object
a_mgr.PaintingsUnderBid[1].BidInProgress = FALSE;
```

Specifying row values

To assign a value to a row, you must use an object reference for an object that is the same class or is a subclass of the row's class. The following example illustrates referencing an attribute in an array:

```
txt_array = Array of TextData = new;
txt_item : TextData = new(value = ‘abc’);
txt_array[1] = txt_item;
```

Adding new array rows

TOOL allows you to add a new row to the array by specifying an object value for a nonexistent row number. You can assign a value to any nonexistent row number. For example, if the last row in the array is 4, you might add a new row 5 or 100 or 999. You can specify an existing object or an object constructor for the object value. If you do not specify the next row in the array (as in the 5th row for the array above), then Forte automatically adds intervening rows to the array, each with a value of NIL.

Processing array rows

To process the rows in an array, you can use TOOL looping statements. In the following example, the **while** statement expression uses the Items attribute for the array object to get the total number of rows in the array.

Example: processing array rows

```
txt_array : Array of TextData = new;
... add rows to txt_array ...
i : integer = 1;
while i < txt_array.Items do
  txt_array[i].Concat(‘extra’);
  i = i + 1;
end while;
```

The **for** statement provides a special syntax for working with an array. In the following example, “txt_array” is an array and the **for** statement repeats the processing statements for each row in the array. “t” is a variable that contains the current array row that is being processed.

Example: using **for** statement with array

```
txt_array : Array of TextData = new;
... add rows to txt_array ...
for t in txt_array do
  -- ‘t’ is automatically declared as TextData
  t.Concat(‘extra’);
end for;
```

Referencing an Attribute in a Row

To reference an individual attribute in an array row, you must specify both the row number and the attribute name.

array_reference[*n*].*attribute*

The following examples illustrate this:

Example: referencing an attribute in an array

```
painters : Array of Artist = new;
painters[1] = new;
painters[1].Born = 1844;
painters[1].Name = new(value = ‘Vincent Van Gogh’);
```

An attribute in an array row is like any attribute

Note that a reference to an attribute in an array row is like any other reference to an attribute. You can use this syntax anywhere you can use an array reference. For example, you could set the value of an attribute in an array row or use in it an expression.

Use this same syntax if the row that the attribute belongs to is itself an attribute of an object or an attribute of array row. For example:

```
-- .. nonsense example, but it illustrates the point ...
a_mgrs : Array of AuctionMgr= new;
a _mgrs[1] = new;
a_mgrs[1].PaintingsUnderBid = new;
-- Allocate array
a_mgrs[1].PaintingsUnderBid[1] = new;
a_mgrs[1].PaintingsUnderBid[1].BidInProgress = FALSE;
```

Assigning a value to an attribute in an array

To assign a value to a single attribute in the row, you can specify any value that is appropriate for the data type of the attribute. For example:

Example: assigning values

```
painters : Array of Artist = new;
painters[1] = new;
painters[1].Born = 1844;
name : TextData = new (value = ‘Vincent Van Gogh’);
painters[1].Name = name;
```

There is no way to assign a value to an entire column in an array. You must assign values to the individual attributes in the column.

Variables

A Forte variable is a name used in TOOL code to refer to a single data item. Every variable has a data type. It can have a simple data type or a class type. If the variable has a numeric, boolean, or string data type, the variable itself contains the data. If the variable has a class type, the variable points to the object or objects that contain the data. Figure 8 illustrates the equivalent of the following code, which assigns a simple string to variable “b” and a class type of Artist to variable “a.”

```
b : string = ‘buzzard’;
a : Artist = new (value = ‘Henri Rousseau’);
```

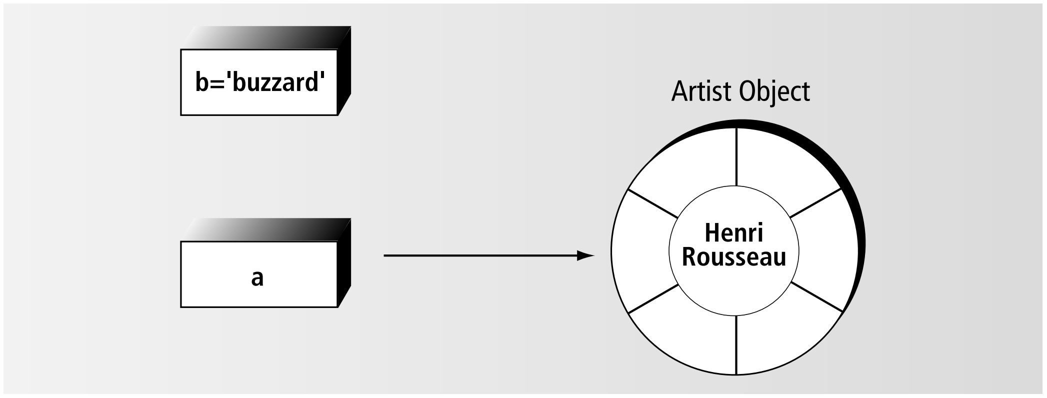


Figure 8 *Simple Variable and Class Type Variable*

You must declare a variable in your TOOL code before you can reference it. After you have declared the variable, you can assign a value to it, reference it, or include it in expressions.

Declaring a Variable

You can declare a variable anywhere within your TOOL code. You must specify the variable name and type. Optionally, you can specify an expression that sets the initial value of the variable.

variable_name [**,** *variable_name*]... **:** *type* [= *expression*]

The scope of the variable is from the point where you declare it until the end of the current statement block. If you declare it at the start of a method, its scope is for the entire method.

Variable name

The variable name identifies the variable for use within the current statement block. It can be any legal TOOL name and must be unique for the current block. If the variable name is the same as the name of a variable declared in an enclosing statement block, the new variable will “hide” the existing variable. The following example illustrates:

Example: name scoping for variables

```
i : integer = 10;
for j in 1 to 5 do
  i : TextData = new(value = ‘start’);
  i.Concat(‘more’);
  ... process ...
end for;
i = i + 5;
```

You can declare multiple variables in a single definition; simply specify more than one variable name. TOOL creates a separate variable for each name, using the same type and initial value. For example:

Example: declaring two variables

```
i, j : integer = 0;
-- Three new objects
t, u, v : TextData = new(value = ‘x’);
```

Variable type

The variable type can be any simple type, any class (or interface), or any array. When selecting the class for a variable, keep in mind the advantage of having the declared type of a variable at the highest level of the class hierarchy as possible. This makes your code more reusable because a single variable can point to many different classes of objects.

Initial value

The expression is the initial value for the variable. This can be any value that is compatible with the data type of the variable. For a variable of a class type, you can use an object or array constructor to specify the initial values for the object or array.

If the variable is a simple data type and you do not specify the initial value, it has the default value for the data type. If the variable is a class type and you do not specify the initial value, it has a default value of NIL.

Referencing a Variable

To reference a variable, use the variable name. The following example illustrates:

Example: referencing a variable

```
i : integer = 0;
j : integer = 10;
j = j + i;
```

You can use a variable to specify a value anywhere in TOOL, as long as the data type meets the requirements of the expression.

Assigning a Variable

To assign a value to a variable, use the following syntax:

variable = *expression*

The expression can be any value that is compatible with the data type of the variable. The following example illustrates:

Example: assigning a value to a variable

```
i : integer = 0;
j : integer = 10;
j = j + i;
```

Named Constants

A named constant is a literal string or numeric value that has a name. To declare a named constant, you specify a constant name and a value. You can then use the name in place of the value in your TOOL code. Forte uses three kinds of named constants: project constants, class constants, and local constants.

Project constants

A project constant is a named constant that can be accessed by any component in the project. You declare project constants using the Project Workshop (see *A Guide to the Forte 4GL Workshops*); you can also use any project constants defined in your supplier plans.

Class constants

A class constant is a named constant that can be accessed by any element in the class. You declare class constants using the Class Workshop (see *A Guide to the Forte 4GL Workshops*). After declaring the class constant, you can use it in your TOOL code.

Local constants

A local constant is a named constant that can be accessed only within the current statement block. You must declare a local constant in your TOOL code before you can reference it. After you have declared the local constant, you use it in the current statement block.

Declaring a Local Constant

You can declare a local constant anywhere in your TOOL code with the **constant** statement. You must specify the constant name and value. The syntax is:

constant *constant_name* = *value*

The scope of the constant is from the point where you declare it until the end of the current statement block. If you declare it at the start of your method, its scope is for the entire method. The following example illustrates declaring a local constant:

```
constant pi = 3.14159268;
constant  SECONDS_PER_HOUR = 3600;
```

Constant name

The constant name identifies the constant for use within the current statement block. It can be any legal TOOL name and must be unique for the current block. If the constant name is the same as the name of a global constant, global variable, or a data item declared in an enclosing statement block, the new named constant will “hide” the existing data item.

Constant value

You declare either a numeric or string value for the named constant. The data type of the value determines the data type of the constant. Once you specify the value for a constant, you cannot change it.

Referencing a Named Constant

To reference a named constant, use the constant name. For example:

Example: referencing a named constant

```
perimeter, radius : double;
radius = 2.0;
constant PI = 3.14159268;
perimeter = 2 * pi * radius;
```

You can use a named constant to specify a value anywhere in the TOOL, as long as the data type meets the requirements of the expression.

If the named constant is a class constant that is not in the current class, other classes must reference the constant with the following syntax:

class_name.constant_name

Using Named Constants in Expressions

Because named constants represent literal values, you can use them in any expression where a literal value is appropriate.

However, because named constants are read-only values, you cannot assign a value to them. This means you cannot pass a named constant as an output parameter or use it on the left side of any assignment statement.

The following example illustrates the use of named constants in an expression:

Example: named constants in expressions

```
perimeter, radius : double;
radius = 2.0;
constant PI = 3.14159268;
perimeter = 2 * pi * radius;
-- following line is ERROR: CANNOT ASSIGN CONSTANT.
PI = 3.14159268;
```

Cursors

A cursor is a row marker used to work with a set of rows retrieved from a database table. You declare cursors for a project using the Cursor Workshop.

In the Cursor Workshop, you define the cursor and give it a name. In order to reference the cursor in your TOOL SQL statements, you must first declare a cursor reference variable using the following format:

variable_name **:** *cursor_name*

Use variable to reference a cursor, not cursor name

From then on, you must use the variable name to reference the cursor, not the cursor name. The syntax diagrams in Chapter 3, “TOOL Statement Reference,” use the term “cursor reference” to indicate that you must use the variable name, not the cursor name.

The exception to this is the **for** statement, where you must use the cursor name, not the cursor reference.

Service Objects

A service object is an object that you reference directly by name. Because the service object is associated with the project as a whole, the scope of the service object's name is the entire project. (See *A Guide to the Forte 4GL Workshops* for information about the purpose of service objects and how to create them.)

You create service objects in the Project Workshop. In your TOOL code, you treat a service object as you would treat any other existing object. Use the service object's name to reference the object.

Example: referencing a service object

```
-- Assume AuctionService and ImageService are defined
a_mgr : AuctionMgr = AuctionService;
i_mgr : ImageMgr = ImageService;
```

Restrictions

Because Forte cannot predict the order in which service objects will be initialized, there are some restrictions when referencing service objects:

1. You cannot reference a service object in the Init method of a class being used as the type for any service object.
2. You cannot reference a service object in a virtual attribute expression of a class being used as the type for any service object.

Because the service object name refers directly to an existing object, you cannot reassign the object associated with the name. However, you can change its attributes. The special service object attributes can be set only in the service object definition, so you cannot change these attributes in your TOOL code. However, you can change any of the attributes defined by the service object's class.

Chapter 3

TOOL Statement Reference

This chapter describes the TOOL statements that you use to write a method, named event handler, or cursor. Statements appear in alphabetical order.

Assignment

The assignment statement sets of the value of a variable or attribute.

Syntax

data_item = *expression*;

Example

```
customer = 543682;
txt_object = other_txt_object;
txt_object = new(value = ‘hello there’);
```

Description

Use the assignment statement to set the value of a data item.

The data item can be a reference to a variable, attribute, parameter, or array row.

The expression specifies the value for the data item. The expression must evaluate to a value that is compatible with the type of the data item.

If the data item has a class type, the expression can be the same class as or a subclass of the data item’s declared class. You can either reference an existing object or use an object constructor to create a new object as the value.

If the data item has an interface type, the expression must be a class that implements the interface (or a subclass of a class that implements the interface).

Begin

The **begin** statement defines a compound statement, which provides local exception handling for a group of statements within a statement block.

Syntax

begin
  [*statement*;]...
  exception
    [**when** *variable_name* : *class* **do** *statement_block*]...
  [**else** [**do**] *statement_block*]
end;

Example

```
begin
  ImageStatusMessage.SetValue(‘Requesting Image.’);
  PaintingImage = self.ImageManager.GetImage
      (name = self.thePaintingForBid.Name);
exception
  when e : GenericException do
    ImageStatusMessage.SetValue(‘Image not available.’);
    task.ErrorMgr.Clear();
end;
```

Description

A compound statement is a set of statements within a statement block that provides its own exception handling. You can use a compound statement anywhere an individual statement is allowed. The **begin** and **end** key words define the compound statement. The **exception** clause provides the exception handling for the compound statement.

If you use a compound statement within a statement block, you can handle an exception without exiting the statement block. A compound statement can contain any TOOL statements, including other compound statements.

Example:
nesting **begin** statements

```
begin
  total_chars : IntegerNullable (isNull = TRUE);
  count : integer = 0;
  begin
    f : File = file_in_param;
    -- File from parameter
    f.Open(SP_AM_READ);
    t : TextData = new;
    while TRUE do
      chars_in_line : integer = f.ReadLine(t, FALSE);
      if chars_in_line < 0 then
        exit;
      end if;
      total_chars.Add(ret, chars_in_line);
      count = count + 1;
    end while;
    f.Close();
  exception
    when e : FileResourceException do
      self.ErrorMgr.ShowErrors(TRUE);
      total_chars.IsNull = TRUE;
      f.Close();
  end;
  total_chars = total_chars / count;
exception
  when e : ArithmeticException do
    task.ErrorMgr.Clear();
    total_chars.IsNull = TRUE;
end;
```

You cannot use the **exit** statement within a compound statement to exit the compound statement. If you do include **exit** in a compound statement, Forte exits from the closest enclosing control statement. If there is no enclosing control statement, this produces an error.

For information on defining the exception handlers for the compound statement, see "Exception" on page 113.

Variables in Compound Statements

A compound statement defines the scope for the variables that you declare within it. A variable that is declared within the compound statement exists from the point it is declared to the end of the compound statement.

The following example illustrates:

Example: variable in compound statement

```
i : integer = 0;
begin
  -- i is an integer here
  i : float = 10.2;
  -- i is a float here.
end;
-- i is an integer again.
i = i + 10;
```

The compound statement also defines the scope for the constants that you declare within it. See "Name Resolution" on page 52 for information about the scopes for variables and constants.

Begin Transaction

The **begin transaction ... end transaction** statement starts a statement block that is executed as a Forte transaction.

Syntax

[*label* :] **begin** [**dependent | independent | nested**] **transaction**
  [**do**] *statement_block*
  [*exception_handler*]
end [**transaction**]**;**

Example

```
begin transaction do
  sql insert into mytab (a) values (10) on session db_s1;
  sql insert into othertab (b) values (12)
    on session db_s2;
exception
  when e : AbortException do
    task.ErrorMgr.ShowErrors(TRUE);
end transaction;
```

Description

The **begin transaction... end transaction** statement defines a Forte transaction. All the statements within the statement block (and all methods invoked from the statement block) are treated as a single unit of work, and either succeed or fail together.

The only exception to this is that new tasks started by the **start task** statement are not part of the transaction unless you use the **start task** statement's **dependent** or **nested** option. See "Transaction Clause" on page 150.

Forte automatically commits the transaction when the statement block is executed successfully (that is, when control leaves the statement block without raising an exception). Forte aborts the transaction if an exception is raised that is caught by the **exception** section for the transaction block or the exception passes through the **begin transaction** statement.

There is no abort statement. To abort the transaction, use the Abort or AbortNested method on the TransactionHandle object referenced by the **transaction** key word. See the Framework Library online Help for information on the TransactionHandle class.

Transactional and non-transactional objects

A transactional object is an object that can participate in a Forte transaction. Changes made to the object during a transaction will be logged so that they can be rolled back if the transaction is aborted. Outside the context of a transaction, a transactional object behaves exactly like any other object. You create a transactional object by defining the class as Transactional in the Class Workshop, and by setting the object's IsTransactional attribute to TRUE (explicitly or by setting the default value of the class to TRUE).

A non-transactional object does not have the IsTransactional attribute set and changes made to that object during a transaction are **not** rolled back.

Scalar variables are not transactional

Local scalar variables (such as integer or string) in a method are never transactional. If you change the value of a scalar variable during a transaction and the transaction is aborted, the value of the variable is not rolled back.

Variables are not transactional; their values are never rolled back. The object that a variable points to may be rolled back, but the value of the variable, that is, which object the variable points to, is never rolled back. For example:

Example: local variables in a transaction

```
o1: TextData = new;
o2: TextData = new;
begin transaction
  o2 = o1;
  raise an exception here
  exception
    exception handling code
end transaction
-- o2 still points to o1. It was not rolled back.
```

Transactional, shared objects

A transactional object may also be shared, that is, both its IsTransactional and IsShared attributes are set to TRUE. In order to access or modify a shared transactional object (either through a public attribute or a method), a transaction acquires a transaction lock on the object in addition to the normal mutex lock that regulates concurrent shared object access. The transaction holds the lock until the transaction ends (either by aborting or committing). When the transaction ends, the lock is released and the next waiting transaction (if any) is granted access.

See the Framework Library online Help manual for more information on the Mutex class, and the *Forte 4GL Programming Guide* for more information on shared and transactional objects.

Transaction Type

When you start a transaction, you should consider how the method that starts the transaction will be invoked. In particular, will the method be invoked from another transaction? If it can be invoked from another transaction, you should decide what effect aborting the current transaction will have on the enclosing transaction. Should aborting the current transaction also abort the enclosing transaction? Or should aborting the current transaction have no effect on the enclosing transaction?

A transaction can be one of three types:

Transaction Type	Description
independent	The method that contains the transaction is not invoked from another transaction, and aborting or committing the work in the independent transaction has no effect on any other transactions. Use the keyword **independent** to make a transaction independent. This type is the default if the transaction is not enclosed in another transaction. You will get an error if the method containing the independent transaction is invoked from another transaction. Although you cannot start more than one independent transaction in a single task, when you are multitasking, you can execute concurrent independent transactions in separate tasks.
dependent	If the method that contains the dependent transaction is invoked from another transaction, the dependent transaction becomes part of the enclosing transaction. Both transactions must succeed together or both will fail. Aborting the dependent transaction automatically aborts the enclosing transaction. Successful work in the dependent transaction is not committed until the enclosing transaction commits. This type is the default for a transaction that is enclosed in another transaction. If the method that contains the dependent transaction is not invoked from an enclosing transaction, the transaction behaves as an independent transaction.
nested	If the method that contains the nested transaction is invoked from another transaction, the enclosing transaction can succeed even if the nested transaction fails. Aborting the work in a nested transaction does not abort the enclosing transaction. However, successful work is not committed until the enclosing transaction commits. Do not use nested transactions when executing SQL statements in database sessions; databases do not support nested transactions.

Multitasking and nested transactions

There is one restriction on using multitasking with nested transactions. A task that is running as a nested transaction cannot use the **start task** statement to start a task with the **dependent** or **nested** option. See "Start Task" on page 147 for information on the **dependent** and **nested** options.

Transaction Statement Block

The statement block for a transaction can include any TOOL statements. However, you must be very careful how you combine methods from the TransactionHandle class with the **begin transaction** statement. The **begin transaction** statement itself invokes TransactionHandle methods, and because these methods are invoked in a particular sequence, you can disrupt this sequence if you use the methods incorrectly. See "Using TransactionHandle Objects" on page 36 for information on this.

Using SQL statements in transactions

If you include SQL statements (or the equivalent methods) in a Forte transaction, Forte automatically starts a database transaction, creating the equivalent of a dependent transaction. Both transactions (the enclosing Forte transaction and the dependent database transaction) must succeed for either transaction to be committed. If either transaction aborts, both transactions abort.

Restrictions on SQL statements

There are a few restrictions on which SQL statements you can include in a transaction:

- Because database vendors do not support nested database transactions, you should not put SQL statements (or methods) in nested transactions.
- You should not use any SQL statements to explicitly control the database transaction (such as rollback, commit, or setautocommit). You can, however, use the Forte Abort method to abort both the Forte and DBMS transactions.
- You should also exclude any SQL statements that your database does not allow within transactions (for example, some systems do not allow DDL statements in transactions).

Forte coordinates distributed transactions

A distributed transaction can access more than one database, more than one database session, or more than one database resource manager. Forte coordinates all commits and aborts. For example, if a transaction accesses more than one database resource manager, and one of the resource managers aborts the transaction, Forte notifies the other resource managers that the transaction is aborted. Likewise, when the transaction commits, Forte signals all resource managers to commit.

Refer to the manual *Accessing Databases* for a more detailed discussion of issues related to two-phase commit and failure during commit of a distributed transaction.

Committing a transaction

To end a transaction, you can use the **exit** statement to end the statement block, use the **return** statement to return from the method containing the statement block, or simply finish the last TOOL statement within the **begin transaction** statement block. In all three cases, the transaction will be committed.

Aborting a transaction

To abort a transaction, invoke the Abort or AbortNested method on the TransactionHandle object using the **transaction** key word (see the Framework Library online Help for information on the TransactionHandle class). Also, if any unhandled exceptions reach the transaction block, the transaction is automatically aborted.

Compound statements in a transaction block

You can use compound statements within the transaction statement block. However, if you raise an exception within the compound statement and it is caught by the compound statement's exception handler and is not passed on, the transaction will not be aborted. In order to abort the transaction from within a compound statement, you must ensure that the exception reaches the **begin transaction** statement's exception handler.

AbortException exception

If the transaction cannot succeed, Forte aborts the transaction and raises an AbortException object (see the Framework Library online Help for information on the AbortException class). If you do not explicitly handle the AbortException exception within the **begin transaction** statement block, Forte automatically handles it for you.

Problems that raise an AbortException object include:

- shared transactional object deadlock
- distributed access exception occurring as a result of a remote message executed as part of the transaction
- some SQL transaction failures that are reported to the Forte transaction manager
- a task participating as a dependent in a transaction is aborted
- an invocation of the Abort method on the TransactionHandle class with the raisedException parameter set to TRUE

End transaction and multitasking

Forte allows multiple tasks to participate in a single transaction (see "Start Task" on page 147). If the task that began the transaction tries to commit the transaction while other tasks are still participating, the beginning task will block, waiting for the other tasks to complete their work. This can occur when the task that began the transaction reaches the **end transaction** clause, tries to return from the method that contains the **begin transaction** statement, or tries to exit from the **begin transaction** statement. A task participating as a dependent in a transaction typically completes its work when the task terminates successfully or invokes the transaction.

Exception Handling

The exception handler for the **begin transaction** statement provides exception handling for the statement as a whole. See “Exception” on page 113 for details about using exception handlers.

Whenever control enters the exception handler for the **begin transaction** statement, the transaction has already been aborted. Therefore, any exception handling you do at this point should be for handling the abort.

Handling the AbortException exception

When the transaction cannot succeed, Forte aborts the transaction and raises the AbortException exception. We recommend that you provide a **when** clause to handle this system-generated exception in the **exception** clause of the **begin transaction** statement. Remember, if you do not explicitly handle the AbortException exception, it will be passed on to the enclosing exception handler like any other exception.

The AbortException should always be handled in the main exception handler for the **begin transaction** statement. If you handle the AbortException exception within any of the code nested within the **begin transaction** statement, you should also use the **raise** statement to ensure that the AbortException reaches the **begin transaction** statement’s **exception** clause. This ensures that the transaction is correctly aborted.

Label

The optional label identifies the **begin transaction** statement for use with the **exit** statement. When you are nesting control statements, you can use the **exit** statement with a **begin transaction** statement label in a statement block, to pass control to the next statement after the statement block. The label name must be unique for the current block (label names share the same name scope as variables and other components). Note that the **exit** statement will commit the transaction.

Case

The **case** statement selects one statement block from a set of alternatives based on the value of an integer expression, and executes the selected statement block.

Syntax

[*label:*] **case** *integer_expression* [**is**]
  [**when** *value* **do** *statement_block*]...
  [**else** [**do**] *statement_block*]
  [*exception_handler*]
end [**case**]**;**

Example

```
t : TextData = new;
line_size : integer = file_from_above.ReadLine(t, FALSE);
case line_size is
  when -1 do
    task.Part.LogMgr.Putline('Past end of file.');
  when 0 do
    task.Part.LogMgr.Putline('Empty line.');
  else do
    task.Part.LogMgr.Putline('Line with characters.');
end case;
```

Description

Forte begins execution by evaluating the **case** expression. Next, it checks each **when** clause to see if there is a value that is equal to the **case** expression. If there is a match, Forte executes the corresponding statement block and then exits the **case** statement. If there is no match, Forte checks for an **else** clause. If there is an **else** clause, Forte executes the **else** statement block and exits from the **case** statement. If there is no **else** clause, Forte simply exits from the **case** statement.

Expression

The **case** expression must be an integer expression. The data type of the corresponding values in the **when** clauses (described below) must also be an integer.

Example: case expression

```
t : TextData = new;
line_size : integer = file_from_above.ReadLine(t, FALSE);
case line_size is
-- line_size must evaluate to an integer
  when 1 do
    ...
end case;
```

When Clause

The **when** clause specifies one integer value that is a possible result of the expression and provides a statement block to be executed for that particular value. The value for each **when** clause must be an integer constant.

If you specify the same value in more than one **when** clause, you will get a compile time error.

Statement Block

The statement block for a **when** clause can include any TOOL statements. You can use an **exit** statement within the statement block to exit from the **case** statement.

Example: use of **exit** statement with **case**

```
case ...
  when 1 do
    if ...condition... then
      exit;
      -- Exits case statement.
    end if;
  when 2 do
    ...
end case;
```

You cannot use the **continue** statement to repeat the **case** statement. If you do include a **continue** statement in the event block, Forte repeats the closest enclosing loop statement (**for**, **while**, or **event loop**). If there is no enclosing loop statement, this produces an error.

Exception Handling

The **exception** clause of the **case** statement provides an exception handler for the **case** statement as a whole. If you wish to provide exception handling for an individual **when** or **else** clause, you must use the **begin** statement within its statement block to define a compound statement. The compound statement can have its own exception handler. See “Begin” on page 93 for information about compound statements.

The Label

The optional label identifies the **case** statement for use with the **exit** statement. When you are nesting control statements, you can use the **exit** statement with a **case** statement label within a statement block to pass control to the next statement after that statement block. The label name must be unique for the current block (label names share the same name scope as variables and constants within the current name scope).

Constant

The **constant** statement declares a named constant, whose scope is from the point you declare it to the end of the block.

Syntax

constant *name* = *value*;

Example

```
constant PI = 3.14159268;
constant seconds_per_hour = 3600;
constant COMPANY_NAME = ‘Forte Software Inc.’;
```

Description

The **constant** statement declares a named constant within a statement block. The scope of the constant is from the point where you declare it until the end of the current statement block.

Constant Name

The constant name can be any legal Forte name and must be unique for the current statement block. Because constants share the same name scope as several other components, if the constant has the same name as a component in an enclosing scope, the new named constant will “hide” the existing component. See “Name Resolution” on page 52 for information on name resolution.

Constant Value

The constant value can be any numeric or string value. The data type of the value determines the data type of the constant.

Once you specify the value for a constant, you cannot change it.

Continue

The **continue** statement is used with the **while**, **for**, or **event loop** statements to repeat a loop.

Syntax

continue [*label*]**;**

Example

```
event loop
  when <print_button>.Click do
    if self.Window.FileSaveDialog(...parms...) = BV_CANCEL
        then
      continue;
    end if;
    begin
      file_to_print.Open(...);
      file_to_print.WriteText(...);
      ...
    end;
  when <quit_button>.Click do
    ...
end event;
```

Description

When the **continue** statement is executed, Forte transfers control to the beginning of the loop. If you do not use a label, the **continue** statement repeats the current loop. If you use a label, the **continue** statement transfers control to the loop statement with the specified label. The following example illustrates:

Example:
continue with a label

```
ev_loop : event loop
  when <save_button>.Click do
    while TRUE do
      begin transaction
        ... save work ...
      exception
        when e : AbortException do
        if self.Window.QuestionDialog('Try Again?') =
            BV_YES then
          exit;
          -- Exits transaction block and retries.
        else
          continue ev_loop;
        end if;
      end transaction;
    end while;
  when ...
    ...
end event;
```

The **continue** statement was designed for use with the **while**, **for**, and **event loop** statements. If you use **continue** in a non-looping statement, Forte transfers control to the closest enclosing loop statement. If there is no enclosing loop statement, this produces an error.

You cannot use the **continue** statement in the **exception** section of the **while**, **for**, or **event loop** statements. When control passes to the exception section, you cannot repeat the loop. To leave the exception section early, you must use the **exit** statement.

Using a Label

When you are nesting loop statements, you can use the **continue** statement with a label to transfer control to a labeled loop statement. Forte forces another iteration of the loop.

Declaration

The declaration statement creates a variable of any type, whose scope is from the point it is declared to the end of the block.

Syntax

variable_name [, *variable_name*]... : *type* [= *expression*];

Example

```
i : integer;
a, b, c : double = 10.0;
x : TextData;
t : TextData = new (value = ‘hello’, IsTransactional = TRUE);
```

Description

You can declare a variable anywhere in TOOL code. You must specify the variable name and type. Optionally, you can use an expression to specify an initial value for the variable.

The scope of the variable is from the point where you declare it until the end of the current statement block.

Variable name

The variable name identifies the variable for use within the current statement block. It can be any legal TOOL name and must be unique for the current block. If the variable name is the same as the name of a variable declared in an enclosing statement block, the new variable will “hide” the existing variable.

To declare multiple variables with a single definition, specify multiple variable names. TOOL creates a separate variable for each name, using the same type and initial value.

Variable type

The variable type can be any simple type, any class, or any interface. See “Simple Data Types” on page 54 for information on the simple data types. The Forte classes are described throughout the Forte documentation set.

Initial value

The expression is the initial value for the variable and can be any value that is compatible with the data type of the variable. For a variable whose type is a class, use an object constructor (or array constructor for the Array or LargeArray classes) to specify the initial value. See Chapter 2, “Language Elements,” for information on object and array constructors.

If the variable is a simple data type and you do not specify the initial value, it has the default value for the data type (zero for numeric types and NIL for the string type). If the variable is a class or interface and you do not specify the initial value, it has a default value of NIL. Note that the DataValue subclasses, including TextData, also have a default value of NIL.

Event

The **event** statement responds to one or more events and has two forms:

- An **event loop** statement waits for any number of events and continues responding to them until you explicitly exit the loop.
- An **event case** statement responds to the first event it receives and then automatically exits the statement.

Syntax

[*label*:] **event loop** [**is**]
  [**preregister** *statement_list*]
  [[**postregister**] *statement_list*]
  [**when** *event_specification* **do** *statement_block*]...
  [*exception_handler*]
end [**event**];

[*label*:] **event case** [**is**]
  [**preregister** *statement_list*]
  [[**postregister**] *statement_list*]
  [**when** *event_specification* **do** *statement_block*]...
  [*exception_handler*]
end [**event**];

event_specification is:

[*object_value*].*event* [(*variable* [**:** *type*] [= *event_parameter*]
  [**,** *variable* [= *event_parameter*]]... [**,** *variable* = **return** | **exception**])]

Example

```
event loop
  when <save_button>.Click do
    self.SaveData();
  when <quit_button>.Click do
    exit;
  when task.Shutdown do
    exit;
end event;
```

Description

The **event** statement provides two different ways for you to respond to events.

The **event loop** statement responds to any number of events and then exits the statement in response to one of the events. It continues waiting for events until the loop ends.

The **event case** statement responds to one, and only one, event and then automatically exits the statement.

The **register** statement (see “Register” on page 131) lets you include a named event handler in either version of the **event** statement.

The following two sections provide more information about the **event loop** and the **event case** statements.

Event Loop Statement

For the **event loop** statement, Forte begins execution by:

1. Executing the **preregister** clause (if provided).
2. Evaluating the event specifications and registering all these events with the current task.
3. Executing the **postregister** clause (if provided).

Forte then waits to receive an event. When one of the events for which the statement has registered enters the task's event queue, Forte executes the statement block for that particular event. After completing the statement block, the **event loop** statement checks the event queue for the next event. If there is more than one event on the queue that the statement has registered for, it uses the one closest to the top of the queue. If there are no events, Forte waits.

Because the event loop continues to wait for events, you must explicitly end the loop with one of the following:

- an **exit** statement
- a **return** statement to exit from the method that contains the event loop
- a labeled **continue** statement to continue an enclosing loop
- a **raise** statement to generate an exception

At least one of the events in the loop should contain one of these statements in its statement block.

Event Case Statement

For the **event case** statement, Forte begins execution by:

1. Executing the **preregister** clause (if provided).
2. Evaluating the event specifications and registering all these events with the current task.
3. Executing the **postregister** clause (if provided).

Forte then waits to receive an event. When one of the events for which the statement has registered enters the task's event queue, Forte executes the statement block for that particular event. After completing the statement block, Forte exits the **event case** statement.

Example:
event case statement

```
tsk_desc : TaskDesc;
tsk_desc = start tasko.meth() where completion = event;
  ... other work while task is executing ...
event case
  when o.meth_return(x = return) do
  ... return value is in x ...
  when o.meth_exception(e = exception, f = errs);
    ... an exception occurred in it ...
end event;
```

Preregister Clause

The optional **preregister** clause provides a list of statements to be executed *before* Forte registers the events in the **when** clause list. The **preregister** clause can include any TOOL statements.

register statement

The **preregister** clause is especially useful for including named event handlers in the event loop. Using the **register** statement in the **preregister** clause is the primary mechanism for including an event handler's code within one or more **event** statements. For example:

```
event loop
  preregister
    -- Include the ArtObjectWindow's event handler in this
    -- event loop.
    register artObjectWindow.artObjectHandler
     (artType = 'Performance');
  -- This window has Post Shutdown attached to the close box.
  when task.Shutdown do
    exit;
...
end event;
```

See “Register” on page 131 for further information about using the **register** statement within an **event** statement.

Forte registers the events from event handlers in the **preregister** clause before registering the events in **when** clauses of the **event** statement. Therefore, if the same event is registered by an event handler in the **preregister** clause and a **when** clause, the **when** clause supersedes the **preregister** clause and the event handling code in the **preregister** clause becomes inactive.

Scoping for variables

The statement list for the **preregister** clause determines the scope for any variables or constants that are declared within it. Variables and constants that you declare within the statement list are available within the **preregister** clause only and cannot be accessed by the rest of the **event** statement.

Exception handling

The statement list in the **preregister** clause does not allow an exception handler; exceptions that occur in the **preregister** clause are handled by the **exception** clause for the **event** statement. See “Exception Handling” on page 122 for information about exception handling for the **event** statement. (Of course, the statement list in the **preregister** clause can include a **begin**/**end** statement with its own exception handler.)

Postregister Clause

The optional **postregister** clause provides a list of statements to be executed *after* Forte registers the events in the **when** clause list, but before the events are handled by the **event** statement. The **postregister** clause can include any TOOL statements.

The **postregister** clause is useful for ensuring that the code that posts an event is always executed after the **event** statement has registered for that event (and so is ready to receive it). The following example shows how you can use the **postregister** clause to synchronize two tasks:

```
event loop
begin
  postregister
    -- StartSale posts an event when it's ready for input.
    start task self.TheSale.StartSale();
  -- When StartSale has posted the event, do something.
  when self.The Sale.ReadyforInput do
    ...start another task related to input...
end event;
```

register statement

You can use the **register** statement in the **postregister** clause to include named event handlers in the **event** statement. Forte registers the events in the **postregister** clause after registering the events in the **when** clauses of the **event** statement. Therefore, if the same event is registered by the **postregister** clause and a **when** clause, the **postregister** clause will supersede the **when** clause. The event handling code in the **when** clause will become inactive. See “Register” on page 131 for further information about using the **register** statement within an **event** statement.

Scoping for variables

The statement list for the **postregister** clause determines the scope for any variables or constants that are declared within it. Variables and constants that you declare within the statement list are available within the **postregister** clause only and cannot be accessed by the rest of the **event** statement.

Exception handling

The statement list in the **postregister** clause does not allow an exception handler; exceptions that occur in the **postregister** clause are handled by the **exception** clause for the **event** statement. See “Exception Handling” on page 111 for information about exception handling for the **event** statement. (Of course, the statement list in the **postregister** clause can include a **begin**/**end** statement with its own exception handler.)

postregister key word

The **postregister** key word is optional when you do not include a **preregister** clause. However, if you include both the **postregister** clause and the **preregister** clause in the same **event** statement, you must use the **postregister** key word.

When Clause

The **when** clause identifies the event that you wish to respond to and provides the corresponding code for that particular event. First, you must specify which event you wish to receive. Second, you can declare a series of variables to receive the parameters that will be passed with the event. Finally, you must provide the statement block to be executed when the event is triggered.

Event Specification

If the event is for the current object, you need only specify the event name. (The “current object” is the object on which the current method, or event handler, is operating.) Otherwise, you must reference the object that will produce the event and specify the event name using dot notation. The event can be any event defined for the object’s class or one of its superclasses.

Example: event specification

```
event loop
  when <save_button>.Click do
    ...
  when self.<quit_button>.Click do
    ...
end event;
```

Event restrictions

You can only register for the method return and exception events from a service object with message dialog duration.

Declaring Variables for Event Parameters

To declare variables to store an event’s parameters, you must enter a variable name for each parameter that you wish to receive. You do not need to specify a data type. Forte automatically uses the data type of the corresponding parameter as it was declared in the original event definition. These variables are always local to the individual **when** clause. You cannot use existing variables in this list. If an event parameter has the same name as an existing variable, the event parameter hides the variable.

To indicate how the variables and parameters correspond, you can use parameter names, parameter positions, or both.

Parameter names

To use parameter names, enter each variable name followed by the corresponding parameter name. These can be in any order and you can exclude any parameters you wish.

Example: declaring variables for event parameters

```
event loop
  when <picture_field_for_drop>.ObjectDrop(dat =
      sourceData, dtype = SourceDataType) do
    if dtype = SD_IMAGE then
      -- Must cast because type of ‘dat’ is Object
      picture_field_for_drop = ImageData(dat);
    end if;
  ...
end event;
```

Parameter positions

To use parameter positions, enter the variable names in the same order as the parameters in the event definition. This order must be identical to the order in the original definition and you cannot exclude any parameters unless they are at the end of the series.

To use a combination of the two techniques, you must first enter the variable names by position. You can then use parameter names for the remaining variables in any order.

Example: parameters by name and position

```
event loop
  when self.Window.Form.AfterMarkLine
      (sx, sy, ey = EndY, ex = EndX) do
    -- ‘sx’ and ‘sy’ are starting coordinates of the line
    -- ‘ex’ and ‘ey’ are the ending.
      ...
end event;
```

Class parameters

Remember that for a parameter whose type is a class, the variable is a reference to the object, not a copy of the object. Therefore, if you change the object within the **event** statement, the changes are visible to any other data items that reference the same object.

Statement Block

The statement block provides the code that is executed when the **event** statement receives the specified event. This can include any TOOL statements. Variables that you declare within the statement block are local to the block.

Nesting **event** statements

You can nest an **event** statement in your statement block either by using another **event** statement or by invoking a method that contains an **event** statement.

A typical example is when you want to open a new window and the Display method for the window contains an event loop. Another example is using the **start task** statement to start a new task and then use the **event** statement to wait for the return or exception events.

Example:
nesting **event** statements

```
event loop
  when <quit_button>.Click do
    exit;
  when <start_task>.Click do
    tsk_desc : TaskDesc;
    tsk_desc = start task o.meth() where completion =
      event;
    event loop
      when o.meth_return(x = return) do
         ...
      when o.meth_exception(e = exception, f = errs);
        ...
      when <cancel_button>.Click do
        tsk_desc.SetCancel();
    end event;
  when <cancel_button>.Click do
    exit;
end event;
```

For each task, Forte registers the events for all **event** statements that have not yet completed. Therefore, when you nest **event** statements, the event queue may contain events for more than one **event** statement. When a given **event** statement is ready to process an event, it checks the event queue for any event that it is prepared to handle. If there is more than one, it uses the event nearest the top of the queue. Any remaining events stay in the queue until the appropriate **event** statement can handle them.

If more than one **event** statement is registered for the same event, the **event** statement that is currently executing when the event reaches the top of the queue is the one that handles the event. Once the event has been handled, it is removed from the queue. Therefore, an event is never handled more than once.

Exception Handling

The **exception** clause of the **event** statement provides an exception handler for the **event** statement as a whole. Once an exception is processed, Forte exits the **event** statement (even if it is an **event loop** statement).

As a result, if an exception occurs in a **when** clause, even if the **when** clause is in an **event loop**, Forte will exit the loop. One solution is to provide exception handling for an individual **when** clause by enclosing the statement block in a **begin**/**end** statement. The compound statement can have its own exception handler. See “Begin” on page 93 for information about compound statements. Another solution is to put the **when** clauses into a separate event handler, and to then use the **exception** clause for that event handler to handle their exceptions.

The following example illustrates how to include an exception handler in a **when** clause:

```
event loop
  -- Standard shutdown sequence.
  when task.shutdown do
    exit;

  -- Create the tables and populate them.
  when <make_tables_button>.Click do
    begin
      status_line = ’Making tables...’;
      ArtistService.MakeTables();
      status_line = ’Tables created.’;
    exception
      when e : DBResourceException do
        task.ErrorMgr.ShowErrors(TRUE);
      when e : AbortException do
        task.ErrorMgr.ShowErrors(TRUE);
    end;

--/ Create the tables and populate them.
  when <drop_tables_button>.Click do
    begin
      status_line = ’Dropping tables...’;
      ArtistService.DropTables();
      status_line = ’Tables dropped.’;
    exception
      when e : DBResourceException do
        task.ErrorMgr.ShowErrors(TRUE);
      when e : AbortException do
        task.ErrorMgr.ShowErrors(TRUE);
    end;
  ...
end event;
```

See WinDB example **Project:** WinDB • **Class:** DataWindow • **Method:** Display

Label

The optional label identifies the **event** statement for use with the **continue** and **exit** statements. You can use **continue** or **exit** to transfer control to a labeled **event loop** statement. You can use **exit** to end a labeled **event case** statement. The label name must be unique for the statement block (label names share the same name scope as variables and other components).

Exception

The **exception** clause provides exception handling for a method, a compound statement, or the **begin transaction**, **case**, **event**, **event handler**, **for**, **if**, and **while** statements.

Syntax

exception
  [**when** *variable _name*:*class* **do** *statement_block*]...
[**else** [**do**] *statement_block*]

Example

```
begin transaction
  ...
exception
  when e : FileResourceException do
    ... clean up from bad file...
  when e : AbortException do
    ... transaction was aborted ...
  else do
    raise;
    -- Pass these on up.
end transaction;
```

Description

Forte executes the **exception** clause only when there is an exception for the current statement. An exception for the current statement occurs in the following circumstances:

- the system generates an exception in a method invoked by the current statement block.

  (Refer to Forte class reference documentation to see which exceptions are raised by methods for a given class.)
- a **raise** statement in the current statement block generates the exception
- the current statement block encloses a statement block that did not handle the exception
- the current statement block invokes a method that did not handle the exception

When the **exception** clause receives an exception, Forte compares the class of the exception to the classes of the variables declared by the **when** clauses. There are several possible outcomes of this comparison:

- If there is one **when** clause variable with a compatible class, Forte executes the corresponding statement block. After the **when** clause is executed, control passes to the statement immediately following the end of the current statement block.

Example:
use of **exception** clause

```
x : array of doubledata;
  ... x gets filled in ...
begin
  count : integer = 0;
  total : double = 0;
  for y in x do
    count = count + 1;
    total = total + y.value;
  end for;
  total = total / count;
exception
  when a : ArithmeticException do
    task.ErrorMgr.ShowErrors(TRUE);
  total = 0;
end;
```

- If there is more than one variable with a compatible class, Forte uses the one that is closest to the exception's class, that is, the lowest down in the class hierarchy.
- If there is no variable with a compatible class and there is an **else** clause, Forte executes the statement block in the **else** clause. This provides general-purpose exception handling for exceptions not specified in a **when** clause. After the **else** clause is executed, Forte closes the current statement block and control passes to the first statement following the statement block.
- If there is no variable with a compatible class and there is no **else** clause, Forte closes the current statement block and delivers the exception to the exception handler in the closest enclosing statement. If there is no enclosing statement with an exception handler, Forte delivers the exception to the invoking method. If no method in the current task handles the exception, Forte terminates the task.

Fatal errors

If the WorstSeverity attribute of the current task's error manager is SP_ER_FATAL, then a fatal exception was raised by this task and the partition is about to terminate. An exception block can catch fatal exceptions and execute some cleanup or recovery step, but the fatal exception will automatically be raised again when control leaves the **exception** clause.

Remember that if a task terminates with an exception and the method that started the task was defined with an exception event, the exception event will be automatically posted so the starting method is notified that the task exited abnormally.

When Clause

The **when** clause declares a new variable to receive the exception and provides a statement block to be executed when the exception is caught.

when *variable_name* **:** *class*
  do *statement_block*

Variable name and class

The variable name specifies the name of the variable to store the object that was raised. The class specifies the type for the variable. The variable type must be the same class or a superclass of the exceptions that you wish to catch. Forte uses the runtime type of the exception to determine which **when** clause to use to handle it. Do not include more than one **when** clause for the same class.

Statement block

The statement block provides the exception handling code that is executed when the **when** clause catches the exception. This code can include any TOOL statements.

Note that you cannot use the **continue** statement in the exception handler to repeat a looping statement. Once control is passed to the exception handler, you cannot continue the loop. To leave the exception handler, you can use one of the following:

- an **exit** statement to pass control to the next statement
- a **return** statement to return from the method
- a labeled **continue** statement to continue an enclosing loop statement

Remember that the exception is a reference to an object, not an object itself. Therefore, if you make changes to the object with the **when** clause, these changes are visible to any other data items that reference the object.

Else Clause

The **else** cause provides a statement block to be executed when there is no **when** clause to handle the exception. This allows you to provide cleanup processing before the statement block or method is closed. The statement block can contain any TOOL statements.

Using **raise** in **when** or **else** clause

You can use the **raise** statement without a value, to pass the current exception to the enclosing statement block or invoking method. This is useful when you want to provide local cleanup processing, but still want the enclosing statement block or invoking method to handle the exception. (See “Raise” on page 129 for an example.)

Do not handle unexpected exceptions

You should not use the **else** clause to provide generic exception handling for unexpected exceptions. In particular, the AbortException and CancelException exceptions should only be handled at the appropriate points in your application, otherwise they will not abort the transaction or cancel the task as they are intended to. The AbortException should be handled in the exception clause of the **begin transaction** statement and the CancelException should be handled in the exception handler of the task’s starting method.

The following example shows incorrect use of the **else** clause. This code handles all exceptions, including the AbortException and CancelException, so that no exceptions are passed to the invoking method.

Example: incorrect use of **else** clause

```
exception
  ...
  else
    task.ErrorMgr.ShowErrors();
end;
```

If it is appropriate to provide recovery for unexpected exceptions, you should raise the exceptions again after handling. The following code fragment illustrates:

Example: correct use of **else** clause

```
exception
  ...
  else
    recovery code
    raise;  -- This re-raises the exception.
end;
```

Exit

The **exit** statement is used with the **case**, **event loop**, **event case**, **for**, **begin transaction**, and **while** statements to close a statement block and pass control either to the statement that follows the block or, if you specify a label, that follows the labeled statement.

Syntax

exit [*label*]**;**

Example

```
exit;
```

Description

When the **exit** statement is executed, Forte closes the current statement block. If you do not specify a label, Forte transfers control to the statement that follows the current block. If you do specify a label, Forte closes the identified statement and passes control to the statement that follows the end of the labeled statement. The **exit** statement allows you to end loops and complete processing within TOOL control statements. The following example illustrates this:

```
event loop
  when <quit_button>.Click do
    if self.Window.QuestionDialog(‘Are you sure?’) = BV_OK
        then
      exit;
    end if;
  when ...
    ...
end event;
```

The **exit** statement was designed to exit from the **case**, **event loop**, **event case**, **for**, **transaction**, and **while** statements. If you use **exit** within other statements (such as the **if** statement or a **begin** statement), Forte exits from the closest enclosing **case**, **event loop**, **event case**, **for**, **transaction**, or **while** statement. If none of these statements are enclosing the **exit** statement, this produces an error. In the case of the **begin transaction** statement, the **exit** statement will commit the transaction.

Using a Label

When you are nesting control statements, you can use a label with the **exit** statement to end a labeled control statement. Forte transfers control to the next statement immediately following the labeled statement. For example:

Example: **exit** with labeled event loop

```
ev_loop : event loop
  when <save_button>.Click do
    while TRUE do
      begin transaction
        ... save work ...
      exception
        when e : AbortException do
        if self.Window.QuestionDialog(‘Try Again?’)
            = BV_OK
            then
          exit;
          -- Exits transaction block and retries.
        else
          exit ev_loop;
        end if;
      end transaction;
    end while;

  when ...
    ...
end event;
```

The label must identify a **case**, **event loop**, **event case**, **for**, **begin transaction**, or **while** statement, and the **exit** statement must be physically nested within the labeled statement.

For

The **for** statement repeats a statement block for each object in an array, each value within a range, each row in a cursor, or each row in the result set of a **select** statement.

Syntax

Objects in an array

[*label* **:**] **for** *variable_name* **in** *array_reference*
  do *statement_block*
  [*exception_handler*]
end [**for**]**;**

Values in a range

[*label* **:**] **for** *variable_name* **in** *first_value* **to** *second_value*
  [**by** *step_expression*]
  do *statement_block*
  [*exception_handler*]
end [**for**]**;**

Rows in a cursor or result set

[*label* **:**] **for** (*storage_reference_list*) **in**
  {**cursor** *cursor_name* [(*placeholder_settings*)] | *sql_select_statement*}
  do *statement_block*
  [*exception_handler*]
end [**for**]**;**

Example

```
j : integer =1;
for i in 1 to 10 do
  j = j * i;
end for;
```

Description

The **for** statement loops through the statement block a set number of times. Different variations of the **for** statement allow you to repeat a statement block based upon the following:

- for each object in the specified array — this is useful for processing the rows in the array.
- for each value within the specified range—this is useful for looping a fixed number of times, such as 1 to 10.
- for each row in the result set of a cursor or a **select** statement—this is useful for processing information that you retrieve from a database table.

At the start of each loop iteration, Forte sets the variable or storage reference list to the current object, current value in the range, or the current row in the result set. When there are no objects left in the array, values in the range, or rows in the result set, Forte exits the statement.

The following sections describe how to use the variations of the **for** statement. This is followed by general information about exception handling.

Using an Array

Declaring the variable

To use an array, you specify a variable name to hold the object being processed. This can be any legal Forte name. Forte automatically declares a new variable, which has a scope of the entire **for** statement. (You cannot use an existing variable.) The variable's type is automatically set to the class of the objects in the array.

The array reference

The array reference can be any expression that references an existing array. (If the array expression evaluates to NIL, the loop will not execute.)

The statement block

The statement block can include any TOOL statements. Use the **exit** statement to transfer control to the statement after the **end for**. Use the **continue** statement to transfer control to the first statement of the statement block and force a new iteration of the statement block. The variable will be assigned the next object in the array.

The following example illustrates using an array in the **for** statement:

Example: using an array

```
vals : Array of TextData = new;
  ... fill in vals somehow...
for v in vals do
  -- Tacks on ‘ value’ to each entry in vals.
  v.Concat(‘ value’);
end for;
```

When execution begins, Forte assigns the first object in the array to the iteration variable and then executes the statement block. Then, after each execution of the statement block, Forte assigns the next object in the array to the variable. As long as there are still objects in the array, Forte executes the statement block again. When there are no more objects left, Forte exits the **for** statement.

Check for NIL rows

If the array contains rows with a value of NIL, they will be assigned to the iteration variable. Therefore, if your array contains NIL rows, you must check for them.

You should not insert or delete rows in the array while it is being used by the **for** statement. You cannot be sure whether the **for** statement will process the inserted or deleted rows.

Using a Range

Declaring the variable

To use a range, you specify a variable name to hold the value being processed. This can be any legal Forte name. Forte automatically declares a new variable, which has a scope for the entire **for** statement. (You cannot use an existing variable.) The variable's type is automatically set to the same type as the range type.

step_expression

To specify a range, you must enter a first value and a second value, using either integers or floating point numbers. The optional **by** clause allows you to specify a step value to be used for calculating the values within the range. If you do not specify a step value, it defaults to 1. The step value can be a positive or negative number. If the step value is positive, the first value must be lower than the second value. If the step value is negative, the first value must be higher than the second value.

The statement block

The statement block can include any TOOL statements. Use the **exit** statement to transfer control to the statement after the **end for**. Use the **continue** statement to transfer control to the first statement of the statement block and force a new iteration of the statement block. The variable will be assigned the next value in the range.

The following example illustrates using a range in the **for** statement:

Example: using a range

```
a : Array of TextData = new;
for i in 1 to 10 do
  a[i] = new();
end for;
```

When execution begins, Forte tests the first value to make sure it is within the range. If it is, Forte assigns the first value to the variable and executes the statement block. Then, after each execution of the statement block, Forte adds the step value to the variable (1 is the default) and compares the variable's new value with the second value. If the step is positive, as long as the variable is less than or equal to the second value, Forte executes the statement block again. When the variable is greater than the second value, Forte exits the **for** statement. If the step is negative, as long as the variable is greater than or equal to the second value, Forte executes the statement block again. When the variable is less than the second value, Forte exits the **for** statement.

Using a Cursor or a Select Statement

You can use the **for** statement with a result set that was returned by either a cursor or a **select** statement.

To use a cursor, use the **cursor** option and specify the cursor name as defined in the project. The **for** statement automatically opens the cursor, fetches the rows one at a time as it goes through the loop, and, when finished, closes the cursor.

To use a **select** statement, use the **sql** option and specify a **select** statement. Do not use the **into** clause in the **select** statement. Using the search criteria in the **select** statement, the **for** statement creates a temporary cursor and opens the cursor before the loop starts. The **for** statement fetches a single row for each loop iteration, and, when finished, closes the temporary cursor.

When execution begins, Forte assigns the column values from the first row in the result set to the storage reference list and then executes the statement block. Then, each time after it executes the statement block, Forte assigns the next row in the result set to the storage reference list. As long as there are still rows in the result set, Forte executes the statement block again. When no more rows are left, Forte exits the **for** statement and closes the cursor.

Implicit transactions

When you use a **for** loop with a cursor name or **sql select** statement, Forte performs an open cursor as part of the initialization of the **for** loop. If no explicit Forte transaction is in effect, this open cursor is enclosed in an implicit transaction, causing the result set to be buffered (the effect of using cursors in implicit transactions is described in *Accessing Databases*). However, the body of the **for** loop is not part of the transaction.

Storage reference list

The storage reference list for the **for** statement declares a list of variables (simple variables or a series of objects) to hold the column values for the row that is being processed. The storage reference list allows you to access the row's values while you go through the loop. These variables also takes the place of the **into** clause of the **select** statement, so you must not use the **into** clause in the **select** statement. Forte automatically declares new variables, which have a scope for the entire **for** statement. (You cannot use existing variables.)

Declaring simple variables

If you declare a storage reference list of simple variables, the variables must correspond *by position* with the columns in the target list of the **select** statement. Simply enter each variable name followed by a simple data type. The name can be any Forte name. The type can be any simple data type that is compatible with the data type of the column (see *Accessing Databases* for information on data type compatibility). The syntax is:

name **:** *simple_type* [, *name* **:** *simple_type*]...

Declaring a class variable

If you declare a storage reference list using a class, the attributes in the object must correspond *by name* with the columns in the target list of the **select** statement. To declare a class variable, simply enter a variable name followed by a class. Forte automatically creates a new object of that class for each row in the result set. The variable name can be any Forte name and the data types of the attributes must be compatible with the corresponding column data types (see *Accessing Databases* for information on data type compatibility). The syntax is:

(*name* **:** *class*)

Using DataValue objects

If you declare a list of variables of the DataValue subclasses (for example, TextData, DateTimeData, and IntegerNullable) in the storage reference list, Forte automatically allocates objects on each iteration of the loop. NULL values fetched from the database will set the IsNull attribute of the corresponding object. This is the only way to get null values from the database. If you do not specify a nullable type and the database returns a null value, you will get a runtime error.

cursor_name

To specify a cursor, enter the name of any cursor that has been defined *but not opened*. (Note that this is really the cursor's name, not a cursor reference as described under "Cursors" on page 89.) Use the optional placeholder list to specify values for placeholders used in the cursor declaration. This list of values must match the placeholder originally defined for the cursor by position.

Example: using a cursor

```
-- Cursor is declared
cursor alltypes_cursor (inttest : integer)
begin
  select intcol, floatcol from alltypes
    where intcol <= :inttest;
end;
-----------------------------------------------------
-- The following code runs the cursor
io : BasicIo = new;
dbsess : DBSession = ... get a session ...
for (i : IntegerNullable, f : DoubleNullable)
  in cursor alltypes_cursor (3) on session dbsess do
    -- A new ‘i’ and ‘f’ object are allocated each time through
    io.WriteLine(‘Start a new row.’);
    io.WriteLine(i);
    io.WriteLine(f);
end for;
```

Specifying a **select** statement

To specify a **select** statement, use the **sql** option with the following syntax for the **select** statement.

select [<u>**all**</u> | **distinct**] {* | *column_list*} **from** *table_name* [**,** *table_name*]...
  [**where** *search_expression*]
  [**order by** *column* [<u>**asc**</u> | **desc**] [, *column* [<u>**asc**</u> | **desc**]]...]
  [**group by** *column_name* [**,** *column_name*]...
  [**having** *search_expression*]
  [**on session** {*session_object_reference* | <u>**default**</u>}];

Caution Do not use an **into** clause in the **select** statement; the storage reference list performs this function.

The **select** statement in the **for** statement accepts vendor-specific extensions, although none appear in the syntax diagram. See the manual *Accessing Databases* for further information about the **sql select** statement and for information about using vendor-specific clauses in SQL statements.

The statement block

The statement block can include any TOOL statements. Use the **exit** statement to transfer control to the statement after the **end for**. Use the **continue** statement to transfer control to the first statement of the statement block and force a new iteration of the statement block. The variables will be assigned the values from the next row in the result set.

When execution begins, Forte assigns the column values from the first row in the result set to the storage reference list and then executes the statement block. Then, after each execution of the statement block, Forte assigns the next row in the result set to the storage reference variables. If you have included objects in the storage reference list, Forte automatically allocates a new object at each iteration. As long as there are still rows in the result set, Forte executes the statement block again. When no rows are left, Forte exits the **for** statement.

Example: using an **sql select** statement

```
painters : Array of Artist = new;
for (a : Artist) in sql select * from artist_table on
session dbsess do
  -- A new Artist object is allocated each time through.
  painters.AppendRow(a);
end for;
  ...
for x in painters do
  x.WriteToLog();
end for;
```

Exception Handling

The **exception** clause of the **for** statement provides an exception handler for the statement block in the **for** statement. See “Exception” on page 113 for information about using an exception handler.

Statement Label

The optional label identifies the statement for use with the **continue** and **exit** statements. If you are nesting control statements, you can use **continue** or **exit** to transfer control to a labeled **for** statement. The label name must be unique for the current statement block (label names share the same name scope as variables and other components).

If

The **if** statement executes a statement block when the specified boolean condition is true.

Syntax

if *boolean_expression* **then**
   statement_block
[**elseif** *boolean_expression* **then**
   statement_block]...
[**else** [**do**]
   statement_block]
[*exception_handler*]
end [**if**]**;**

Example

```
io : BasicIO = new;
cmd_length : integer;
t : TextData = new;
cmd_length = f.ReadLine(t,TRUE);
if cmd_length < 0 then
  io.WriteLine('Past end of line.');
elseif cmd_length = 0 then
  io.WriteLine('Empty line.');
else
  io.WriteLine('Line is empty.');
end if;
```

Description

Forte begins execution of the **if** statement by evaluating the first boolean expression. If the first expression is true, Forte executes the first statement block and then exits the **if** statement. If the first expression is false, Forte evaluates the **elseif** expressions one at a time until it finds one that is true. Forte then executes the corresponding statement block and exits the **if** statement.

If none of the **elseif** expressions are true or if there are no **elseif** expressions, Forte checks for an **else** statement block. If one is present, Forte executes the **else** statement block and then exits the **if** statement. If there is no **else** statement block, Forte simply exits the **if** statement. (Please notice that there is no space in the **elseif** keyword!)

The simplest version of the **if** statement specifies one condition to be tested and one statement block to execute if the expression is true. For example:

Example: simple **if** statement

```
i : integer = 10;
if i > 5 then
  io.WriteLine('Passed successfully.');
end if;
```

More complex versions of the **if** statement specify a series of conditions to be tested, one after the other (as in the example at the beginning of this section).

The **else** clause specifies a statement block to be executed when all the conditions are false. This is also illustrated in the example at the beginning of this section.

Boolean Expressions

The expressions in the **if** statement must be boolean expressions (described under "Boolean Expressions" on page 56). These can include boolean variables, constants, attributes, and methods that produce a boolean return value.

Example: boolean expression

```
if ((i > 10) and (i < 100)) or not (j > 4) then
  ....
end if;

txt : TextData = new(value = ‘abc’);
if txt.IsEqual(‘abc’) then
  -- ...will happen, because BooleanNullable will pass test...
end if;

if transaction.IsActive then
  -- ... in an active transaction ...
end if;
```

Statement Blocks

The statement blocks in an **if** statement can include any TOOL statements. Note, however, that you cannot use the **exit** or **continue** within an **if** statement to close or repeat the **if** statement blocks. If you do include **exit** or **continue** in an **if** statement block, control passes to the closest enclosing control statement. If there is none, this produces an error.

Exception Handling

The **exception** clause of the **if** statement provides an exception handler for the **if** statement as a whole. If you wish to provide exception handling for an individual **then**, **elseif**, **or else** clause, you must use the **begin** statement within its statement block to define a compound statement. The compound statement can have its own exception handler. See "Begin" on page 93 for information about compound statements.

Method Invocation

Invoking a method executes the method using the values you specify for the parameters.

Syntax

object_reference.*method*[([*name* =] *expr* [, [*name* =] *expr*]...)];

Example

```
t : TextData = new(value = ‘abc’);
t.Concat(source = ‘def’);
  ... or ...
t.Concat(‘def’);
```

Description

To invoke a method, use an object reference to identify the object and then use dot notation to specify the method name. You must also specify a value for each required input parameter.

Object reference

The object reference identifies the object on which you wish to invoke the method and on which the method will operate. If the method is for the current object, you do not need to include the object reference. (The “current object” is the object on which the current method is operating.) If you specify a method name by itself, TOOL assumes the current object.

Method name

The method name identifies the particular method to invoke. This can be any method defined or inherited by the object’s class. If there is more than one method with the same name, Forte checks for method overloading at compile time. Then, at runtime, Forte checks for method overriding. See “Invoking Methods” on page 73 for more information on this.

Parameters

The parameters for a method can be defined for input only, for input-output, or for output only. A method may have no input parameters, or it may have required parameters, optional parameters, or a combination of both.

Required parameters

A parameter is required if the parameter definition does not specify an initial value. If the method has required parameters, you must specify values for all the required parameters when you invoke the method.

To indicate how the values and parameters correspond, you can use the parameter names, the parameter positions, or both.

Parameter names

To use the parameter names, enter each parameter name followed by the value. These can be in any order. Using parameter names rather than positions makes code easier to maintain.

Parameter positions

To specify the parameters by position, enter the values in the same order as the corresponding parameters in the method definition. This order must be identical to the parameters in the original definition.

To use a combination of the two techniques, you must first enter the values by position. You can then use parameter names for the remaining values in any order.

Optional parameters

A parameter is optional if the parameter definition specifies an initial value. If the method has optional parameters, you can use or ignore any of the optional parameters. If you decide to ignore all the parameters and there are no required parameters, you can omit the parentheses when you invoke the method.

If you use parameter names, you can omit any of the optional parameters. If you use parameter positions, you can only omit optional parameters that are at the end of the series.

Parameter values

The value for an input parameter can be any expression that is compatible with the parameter's data type. The value for an output or input-output parameter must be an attribute or variable of the appropriate type.

Strings and TextData parameters

If the method is defined with a TextData parameter, you can specify a simple string as the value when you invoke the method. Forte automatically creates the TextData object, using the string for its TextValue attribute.

Output Parameters

A method may have no output or input/output parameters, or one or more. The value for any output parameter (either input-output or output only) must either be a variable or an attribute. Because the method assigns the output value to the output parameter, the input value you specify must be a variable or attribute that can be on the left side of an assignment statement.

Note that Forte passes output parameters by value result. This means that the value of the parameter is not changed until *after* the method returns.

Class Parameters

If a parameter whose type is a class is for input only, you can pass a reference to an object of the same class or a subclass of the parameter's declared class. (Note that when the declared class of the parameter is different than its runtime class, you may need to cast it before you reference it within the method. See "Casting" on page 69).

However, if a parameter whose type is a class is for input-output or for output only, you must pass a reference to an object of the same class as the parameter's declared class. Anything else is illegal.

Note

Because you are passing a *reference* to an object, not the object itself, even if a parameter is for input only, if the method makes changes to the object, these changes are reflected when you return from the method. This is because both the invoking method and the invoked method are referencing the same object.

If the parameter was defined using the **copy** option, TOOL passes a reference to a copy of the object (see *A Guide to the Forte 4GL Workshops* for information about using the **copy** option when defining method parameters.)

Post

The **post** statement triggers an event. Forte automatically notifies all the currently executing tasks that have registered for the event that the event was posted.

Syntax

post [*object_reference*]**.***event* [([*parameter_name* =] *expression*
[, [*parameter_name* =] *expression*]...)]**;**

Example

```
post self.PaintingAdded (painting = paintingToAdd);
```

Description

The **post** statement triggers an event of any type and optionally, specifies values for the event's parameters. When an event is posted, Forte notifies all tasks and programs registered to receive that event. Each task then queues the event so the relevant **event** statement can respond to it.

The following example shows the posting of the event PaintingAdded, with one event parameter that tells which painting was added.

Example: **post** statement with **event loop**

```
-- In method AddBid on AuctionMgr
begin
  ...
  post self.PaintingAdded (painting = paintingToAdd);
end method;

-- (then define service object AuctionService on AuctionMgr class)

-- In method Display for PaintingListBatch
begin
event loop
  ...
  when AuctionService.PaintingAdded (addedPainting =
      painting) do
    self.PaintingData.AppendRow(addedPainting);
    self.CurrentRow = self.PaintingData.Items;
  ...
end event;
end method;
```

Specifying the Event

To identify the event to be posted, reference the object that is producing the event and specify the event name using dot notation. The event can be any event defined for the object's class.

If no object reference is used, Forte posts the event for the current object.

Specifying the Parameters

When you post an event, the values you specify for the event's parameters must correspond one to one with the parameters defined for the event. For each parameter, you can specify any value that is compatible with the parameter's data type.

To indicate how the values and parameters correspond, you can use the parameter names, the parameter positions, or both.

To use the parameter names, enter each parameter name followed by the value. These name and value pairs can be in any order and you can omit any optional parameters.

Example:
using parameter names

```
name : TextData = new(value = ‘Christy’);
post self.BidCompleted (newValue =
    DecimalData(value = 123.45),
    timeOfBid = DateTimeData(), whoBid = name);
```

To specify the parameters by position, enter the values in the same order as the corresponding parameters in the event definition. This order must be identical to the parameters in the original definition and you cannot exclude any optional parameters unless they come at the end of the series.

Example:
using parameter positions

```
name : TextData = new(value = ‘Christy’);
post self.BidCompleted (DecimalData(value = 123.45),
    DateTimeData(), name);
```

To use a combination of the two techniques, you must first enter the values by position. You can then use parameter names for the remaining values in any order.

Strings and
TextData parameters

If the event was defined with a TextData parameter, you can specify a simple string as the value for the parameter when you post the event. Forte automatically creates the TextData object, using the string for its TextValue attribute.

No derived C data types

Note that event parameters cannot be derived C data types, such as structs or C-style arrays. See *Integrating with External Systems* for information about the C data types.

Raise

The **raise** statement generates an exception to be handled by an **exception** clause. The exception is an object of any class.

Syntax

raise [*exception_object_reference*]**;**

Example

```
ex : GenericException = new;
ex.SetWithParams(DB_ER_ERROR, ‘Bad value %1.’,
    TextData(value = ‘xxx’));
task.ErrorMgr.AddError(ex);
raise ex;
```

Description

When the **raise** statement is executed, Forte delivers the exception to the closest exception handler. The following example illustrates:

Example: **raise** statement and exception handler

```
begin
  ...
  if ... unusual condition ... then
  -- Assume MyExceptClass a subclass of GenericException
    m : MyExceptClass = new;
    raise m;
  end if;
  ...
exception
when m : MyExceptClass do
    -- This block will be executed from ‘raise m’ above.
    exit;
  else
    task.ErrorMgr.ShowErrors(TRUE);
end;
```

Forte starts by checking the current statement (or current compound statement) for an exception handler. If there is no exception handler or if the exception handler cannot handle the exception, Forte searches through the enclosing statements for an exception handler that is prepared to handle it. If the current method cannot handle the exception, Forte delivers the exception to the invoking method.

Example: exception delivered to invoking method

```
-- Declare two methods
myclass.meth1
begin
  self.meth2();
exception
  when m : MyExceptClass do
    -- This block will be executed from the ‘raise m’ in
meth2.
    exit;
  else
    task.ErrorMgr.ShowErrors(TRUE);
end method;
myclass.meth2
begin
  if ... unusual condition ... then
    -- Assume MyExceptClass is declared as subclass of
GenericException
    m : MyExceptClass = new;
    raise m;
  end if;
  ...
end method
```

Caution You should ensure that every exception you raise will be handled by the appropriate exception handler. If the exception is not handled by any of the executing methods in the current task, Forte terminates the task.

Identifying the Exception

Unless you are raising the current exception from an **exception** clause (described below), you must identify the exception you wish to raise. The exception is a reference to an object of any Forte exception class or custom exception class. For your own exceptions, you can define a subclass of the UserException class (see the Framework Library online Help for information on this), or use any class.

Note that the exception you raise is a reference to an object, not the object itself. Therefore, if the exception handler that handles the exception changes the object, the changes are visible to any other data item that references the object.

Raising the Current Exception

After handling an exception in an **exception** clause, you may wish to pass the exception on to an enclosing statement or to the invoking method. This is useful if you want to catch the exception and perform local cleanup processing, but you also want the enclosing statement or invoking method to handle the exception. (This is used most often in the **else** clause of the **exception** clause.) In this case, you can use the **raise** statement without an exception reference because Forte assumes the current exception.

Example: raising the current exception

```
begin
  f : File = ... file from somewhere ...
  f.Open(SP_AM_READ);
  ...other file activity ...
exception
  when e : GenericException do
    f.Close();
    -- Close the file, but pass exception on.
    raise;
end;
```

The only time the object reference is optional with the **raise** statement is when you use it within the **when** or **else** clause of the exception handler.

Register

The **register** statement includes a named event handler in the current **event** statement.

Syntax

[*data_item* =] **register** *event_handler_reference* [(*parameter_list*)]**;**

Example

```
event loop

  preregister
    -- Include the ArtObjectWindow's event handler in this
    -- event loop.
    register artObjectWindow.artObjectHandler
     (artType = 'Performance');

  -- This window has Post Shutdown attached to the close box.
  when task.Shutdown do
    exit;
...
end event;
```

Description

The **register** statement includes an event handler at any point within an **event** statement or another event handler. The only way to add event handler code within a method is to use the **register** statement inside an **event** statement.

The **register** statement is useful in:

- the **preregistration** clause of the **event** statement or event handler definition
- the **postregistration** clause of the **event** statement or event handler definition
- the statement block of a **when** clause in an **event** statement or an event handler definition
- a method invoked from within an **event** statement or event handler definition

You must use the **register** statement within an **event** statement. If no **event** statement is active when the **register** statement is executed, you will get a runtime error.

When the **register** statement is executed, Forte registers all the events specified in the event handler. Registering an event means notifying the object that will be posting the event that the current object is prepared to handle the event. This ensures that the current object will be notified when the event is actually posted.

After the events are registered, the event handler waits, as part of the **event** statement that contains it, to receive an event. When one of the events for which the event handler has registered enters the task's event queue, Forte executes the statement block that the event handler has provided for that particular event. For example:

```
event loop

  preregister
    -- Include the ArtObjectWindow's event handler in this
    -- event loop.
    register artObjectWindow.artObjectHandler
     (artType = 'Performance');

  -- This window has Post Shutdown attached to the close box.
  when task.Shutdown do
    exit;
...
end event;

-- This is the event handler that was registered in the
-- above event loop. Its when clauses now act as if
-- they are part of the above event loop.
event handler ArtObjectWindow.ArtObjectHandler
  (input artType: string)
begin
  preregister
    artobject_data.object_type.SetValue(artType);

  -- Validate the object_type field.
  when <artobject_data.object_type>.AfterValueChange do
    retval : TextData = self.artobject_data.ValidateType();
    if retval != NIL then
      -- Error in validation.  Print message and get rid
      -- of pending events.
      self.Window.MessageDialog(messageText = retval);
      self.Window.PurgeEvents();
    end if;
  -- Validate the whole thing.
  when <artobject_data>.AfterValueChange do
    retval : TextData = self.artobject_data.ValidateAll();
    if retval != NIL then
      -- Error in validation.  Print message and get rid
      -- of pending events.
      self.Window.MessageDialog(messageText = retval);
      self.Window.PurgeEvents();
    end if;
end event;
```

See NestedWindow example **Project:** NestedWindow • **Class:** ClassArtObjectWindow • **Method:** Event Handler

After the **event** statement that contains the **register** statement completes execution, the event handlers that were registered within the **event** statement are automatically unregistered. If you wish to cancel registration of an event handler before the **event** statement completes execution, you can do so using the EventRegistration object.

EventRegistration object

The **register** statement returns an object of type EventRegistration, which identifies the event handler registration that was just performed. You can use the EventRegistration object to cancel the registration of the event handler before the **event** statement completes execution. To cancel registration, invoke the DeregisterHandler method on the EventRegistration object. For example:

Example: using the EventRegistration object

```
-- This method fragment shows how to use the
-- EventRegistration object to register and unregister
-- events. In this example, two nested windows are being
-- swapped in and out of a single parent window.
evReg : EventRegistration = NIL;
artistWindow : ArtistWindow = new;
artobjectWindow : ArtObjectWindow = new;

self.Open();

event loop
  -- Start with an Artist Window
  artistWindow.Window.Row = 3;
  artistWindow.Window.Column = 1;
  artistWindow.Window.Parent = <main_grid>;
  artistWindow.Window.FrameWeight = W_NONE;

  evReg = register artistWindow.artistHandler;

  when task.Shutdown do
    exit;

  -- Launch an ArtistWindow, with a new object.
  when <start_artistwindow_button>.Click do
    evReg.DeRegisterHandler;
    artObjectwindow.close;
    artObjectwindow.window.Parent=nil;
    -- First set up the inner window and it’s data.
    artistWindow = new;
    artistWindow.Window.Row = 3;
    ...
    evReg = Register artistWindow.artistHandler;

  -- Launch an ArtObjectWindow.
  when <start_artobjectwindow_button>.Click do
    artistwindow.close;
    artistwindow.Window.parent = nil;

    evReg.DeRegisterHandler;

    -- First set up the inner window and it’s data.
    artobjectWindow = new;
    artobjectWindow.Window.Row = 3;
    ...
  evReg = Register artObjectWindow.ArtObjectHandler;

end event;
self.Close();
```

Event registration stack

Because you can have more than one **register** statement within the same **event** statement or event handler, it is possible to register the same event more than once. When there is more than one registration for the same event, the event registrations are added to a stack, and the most recent registration supersedes the others. Only the top-most registration in the stack is active; all other registrations are inactive and are ignored when the event is handled.

If the DeregisterHandler method cancels the most recent registration in the stack, the next registration in the stack becomes active. See the Framework Library online Help for information about the EventRegistration class, the DeregisterHandler method, and the event registration stack.

Unless you explicitly unregister it, the event handler stays registered for the duration of the **event** statement. After the **event** statement completes execution, the event handler is automatically unregistered.

Note The **register** statement must be executed on the same partition as the **event** statement that contains it. If the **register** statement is on a remote partition, you will get a runtime error. This situation could occur, for example, if an **event** statement invokes a remote method that contains a **register** statement.

Event Handler Reference

You can register any event handler defined for or inherited by the current object's class. (The "current object" is the object on which the current method, or event handler, is operating.) To reference an event handler for the current object, simply specify the event handler name. For example:

```
register artObjectHandler(artType = 'Performance');
```

Object reference

You can also register any public event handler on any other local object. Specify the object reference with the event handler name using dot notation. The object reference identifies the object for which you wish to register the event handler. The event handler will handle events registered by that handler. For example:

```
-- Before registering the handler, instantiate the object.
artObjectWindow : ArtObjectWindow = new;
-- ...later, in the event loop...
    register artObjectWindow.artObjectHandler
     (artType = 'Performance');
```

The object on which you register the event handler must be located on the same partition as the enclosing **event** statement. If it is not, you will get a runtime error.

Current object for **register** statement

Like invoking a method on an object, registering an event handler on an object changes the "current object." When you register an event handler on an object, that object becomes the current object as long as the event handler executes. Thus, any references to the current object (or "self") within the event handler refer to the object on which the **register** statement was executed.

Use **super** to register an overridden event handler

When you override an event handler, you replace an inherited event handler with a new event handler with the same name. If you need to register the inherited event handler (that you are overriding) in the newer event handler, simply use the **super** key word before the method name to specify the inherited event handler rather than the current event handler.

Example: Overriding an event handler

```
-- Registering handlers from both self and super in an event loop.
event loop
    preregister
       register self.ArtistHandler();
       register self.ResetHandler();
       -- Use the inherited event handler to handle exit button
events.
       register super.ExitHandler();
end event;
```

See InheritedWindow example

Project: InheritedWindow • **Class:** ArtistDataEntryWindow • **Method:** Display

To see this example in context, see the Forte example program InheritedWindow.

Registering an inherited event handler is particularly useful when you are writing a new event handler that adds functionality to an inherited event handler. By registering the superclass's event handler, you include all its functionality within the current event handler. This is similar to invoking "**super**.*method*" in a method that is overriding an inherited method.

Parameter List

Event handler parameters

If the event handler has parameters, you must pass the event handler the information that it requires. The values that you specify when you register the event handler must correspond one to one with the parameters in the event handler definition. You can specify any value that is compatible with the parameter's data type.

Example: event handler parameters

```
register artObjectWindow.artObjectHandler
  (artType = 'Performance');
-- another example
register artistWindow.artistHandler
  (artistName = name, artistCountry = country);
```

Scalar parameters you specify with the **register** statement are copied to the event handler when the **register** statement is executed. If any changes are made to a simple variable *after* it is passed as a parameter to an event handler, the changes are not reflected in the event handler. However, for class parameters, Forte passes a reference to an object, not the object itself. Therefore, if any changes are made to an object after the reference to it is passed as a parameter to the event handler, the changes are reflected in the event handler.

Strings and TextData parameters

If the event handler was defined with a TextData parameter, you can specify a simple string as the value when you register the event handler. Forte automatically creates the TextData object, using the string for its TextValue attribute.

Like methods, event handlers have required parameters, optional parameters, or a combination of optional and required parameters.

Required parameters

A parameter is required if the parameter definition does not specify an initial value. If the event handler has required parameters, you must specify values for all the required parameters when you register the event handler.

To indicate how the values and parameters correspond, you can use the parameter names, the parameter positions, or both. The syntax is:

[*name* =] *value* [, [*name* =] *value*. . .]

Parameter names

To use the parameter names, enter each parameter name followed by the value. These can be in any order. We recommend using parameter names rather than positions because it makes your code easier to maintain.

To specify the parameters by position, enter the values in the same order as the corresponding parameters in the event handler definition. This order must be identical to the parameters in the original definition.

Example:
using parameter positions

```
register artistWindow.artistHandler(name, country);

-- The corresponding event handler definition:
event handler ArtistWindow.ArtistHandler
  (input artistName = string,
   input artistCountry = string)
```

To use a combination of the two techniques, you must first enter the values by position. You can then use parameter names for the remaining values in any order.

Optional parameters

A parameter is optional if the parameter definition specifies an initial value. If the event handler has optional parameters, you can use or ignore any of the optional parameters. If you decide to ignore all the parameters and there are no required parameters, you can omit the parentheses when you register the event handler.

If you use parameter names, you can leave out any of the optional parameters. If you use parameter positions, you can only leave out optional parameters that are at the end of the series.

Class parameters

Because an event handler's class parameter is for input only, you can pass a reference to an object of the same class or a subclass of the parameter's declared class. (Note that when the declared class of the parameter is different than its runtime class, you may need to cast it before you reference it within the event handling code. See "Casting" on page 69.)

Note

Because you are passing a *reference* to an object, not the object itself, even if a parameter is for input only, if the method makes changes to the object, these changes are reflected when you return from the method. This is because both the invoking method and the invoked method reference the same object.

If the parameter was defined using the **copy** option, TOOL passes a reference to the copy of the object (see *A Guide to the Forte 4GL Workshops* for information about the **copy** option for event handler parameters).

Return

The **return** statement returns control from the current method to the invoking method. If the current method is defined with a return type, the **return** statement also returns the return value.

Syntax

return [*value*]**;**

Example

```
return;
return 10;
```

Description

When the **return** statement is executed, Forte returns control to the invoking method, immediately following the point where the current method was invoked. You can use the **return** statement anywhere in your TOOL code to end the current statement block and exit from the method. A **return** statement in the start method for the application exits the application.

Forte ignores any statements in the current statement block that follow the **return** statement. Therefore, the **return** statement should be the last statement in the block.

Return in asynchronous methods

If the **return** statement appears in a method that is invoked asynchronously with the **start task** statement, Forte terminates the task and posts the return event if the **start task** statement requested one. This is because a task cannot return to the method that invoked it. If you started an independent transaction with the **start task** statement, the **return** statement ends the transaction successfully.

Return Value

Return required for method with return type

If a method is defined as having a return type, you must include the **return** statement and specify a return value. This value must be compatible with the method's return type. If you do not specify a return value, this produces a compile time error. See "Writing Methods" on page 22 for information on writing a method.

For example:

Example: return value

```
myclass.mymeth : TextData
begin
  t : TextData = new(value='return value');
  return t;
end method;
```

SQL Close Cursor

The **sql close cursor** statement closes a cursor that was opened with the **sql open cursor** statement.

Syntax

sql close cursor *cursor_reference***;**

Example

```
sql close cursor dbcursor;
```

Description

For a complete description of the **sql close cursor** statement, refer to the manual *Accessing Databases*.

SQL Delete

The **sql delete** statement removes rows from a table.

Syntax

sql delete from *table_name*
  [**where** {*search_condition* | **current of** *cursor_reference*}]
  [**on session** {*session_object_reference* | **default**}]**;**

or

{*numeric_attribute* | *numeric_variable*} = (*sql_delete_statement*);

Example

```
sql delete from artist_table where born < 1500;
```

Description

For a complete description of the **sql delete** statement, refer to the manual *Accessing Databases*.

SQL Execute Immediate

The **sql execute immediate** statement executes one SQL statement.

Syntax

sql execute immediate {*string_literal* | *string_variable* |*attribute* |
TextData_reference} [**on session** {*session_reference* | **default**}]**;**

or

{*numeric_attribute* | *numeric_variable*} =
(*sql_execute_immediate_statement*);

Example

```
sql execute immediate ‘create table xyz (co1 int)’
  on session dbsess;
```

Description

For a complete description of the **sql execute immediate** statement, refer to the manual *Accessing Databases*.

SQL Execute Procedure

The **sql execute procedure** statement executes a database procedure.

Syntax

sql execute procedure *procedure_name*
[([**input** | **output** |**input output**] *parameter* = *expression*
[**,** [**input** | **output** |**input output**] *parameter* = *expression*]...)]
[**on session** {*session_object_reference* | **default**}]**;**

or

DataValue_reference = (*sql _execute_procedure_statement*);

Example

```
empid : integer = 12345;
salaryIncrement : integer = 15000;
-- Passing parameters by position.
sql execute procedure updateSalary(empid, salaryIncrement);

-- Passing parameters by name.
sql execute procedure updateSalary(AddToSalary = salaryIncrement,
   Id = empid);
```

Description

For a complete description of the **sql execute procedure** statement, refer to the manual *Accessing Databases*.

SQL Fetch Cursor

The **sql fetch cursor** statement allows you to fetch the next row, the next *n* rows, or the entire result set.

Syntax

sql fetch [[**next** {*integer_constant* | :*integer_varName*}] **from**]
cursor *cursor_reference*
[**into** {**:***object_reference* | *simple_list* | *array_reference*}]**;**

or

{*numeric_variable* | *numeric_attribute*} = (*sql_fetch_cursor_statement*)**;**

Example

```
sql fetch cursor dbcursor into :a;
```

Description

For a complete description of the **sql fetch cursor** statement, refer to the manual *Accessing Databases*.

SQL Insert

The **sql insert** statement adds a new row to a database table.

Syntax

sql insert into *table_name* [(*column* [**,** *column*]...)]
  {**value**s ({*object_reference* | *simple_list*}) | *select_statement*}
  [**on session** {*session_object_reference* | **default** }]**;**

or

{*numeric_attribute* | *numeric_variable*} = (*sql_insert_statement*);

Example

```
a : Artist = new ... fill in data...;
sql insert into artist_table
  (name, country)
  values (:a.name, :a.country)
  on session dbsess;
```

Description

For a complete description of the **sql execute insert** statement, refer to the manual *Accessing Databases*.

SQL Open Cursor

The **sql open cursor** statement selects rows from a database table to be used with a database cursor.

Syntax

sql open cursor *cursor_reference*
  [(*expression* [, *expression*]...)]
  [**on session** {*session_object_reference* **|** **default**}]**;**

Example

```
dbcursor : empcursor;
empid : integer;
sql open cursor dbcursor (empid) on session dbsess;
```

Description

For a complete description of the **sql open cursor** statement, refer to the manual *Accessing Databases*.

SQL Select

The **sql select** statement retrieves one or more rows from one or more database tables.

Syntax

sql select [**all** | **distinct**] {* | *column_list*}
  [**into** {*object_reference* | *simple_list*}]
  [**from** *table_name* [**,** *table_name*]...]
  [**where** *search_expression*]
  [**group by** *column_name* [, *column_name*]...]
  [**having** *search_expression*]
  [**order by** *column* [**asc** | **desc**] [, *column* [**asc** | **desc**]]...]
  [**on session** {*session_object_reference* | **default**}]**;**

or

{*numeric_attribute* | *numeric_variable*} = (*sql_select_statement*);

Example

```
sql select * into :artist_object from artist_table
where name = :vname
on session dbsess;
i : integer;
vname, vcountry : TextData = new;
i = (sql select name, country into :vname, :vcountry from
      artist_table on session dbsess);
```

Description

For a complete description of the **sql select** statement, refer to the manual *Accessing Databases*.

SQL Update

The **sql update** statement changes values in one or more rows from a database table.

Syntax

sql update *table_name* **set** *column* = *expr* [**,** *column* = *expr*]...
  [**where** {*search_expression* **|** **current of** *cursor_reference*}]
  [**on session** {*session_object_reference* **|** **default**}]**;**

or

{*numeric_attribute* **|** *numeric_variable*} = (*sql_update_statement*);

Example

```
sql update artist_table set born = :vborn
  where name = :vname on session dbsess;
```

Description

For a complete description of the **sql update** statement, refer to the manual *Accessing Databases*.

Start Task

The **start task** statement begins a new task by invoking an asynchronous method.

Syntax

[*data_item* =] **start task** [*object_reference*.]*method* [(*parameter_list*)]
[**where** *setting* [, *setting*]...];

setting is:

completion = {**event** | **<u>ignore</u>**} |
transaction = {**dependent** |**independent** | **nested** | **<u>none</u>**} |

Example

```
start task self.theBid.StartBid
    (bidderName = self.theUserName)
    where completion = event, transaction = dependent;
```

Description

The **start task** statement starts a new task by invoking the specified method asynchronously. Normally, when you invoke a method, the invoking method waits until the invoked method completes. However, when you invoke a method using the **start task** statement, both methods are executed in parallel.

Return value for **tas**k statement

The **start task** statement optionally returns an object of the TaskDesc class. You can use this object to get access to the new task while it executes. For example, you can use the SetCancel method of the TaskDesc class to cancel the new task any time during its execution. See the Framework Library online Help for information on the TaskDesc class.

By default, the method invoked by the **start task** statement does not notify the invoking task when it has completed or that it was terminated. However, if the method has been defined with return and exception events (see “Completion Clause” on page 148), you can use the **completion** option of the **start task** statement to request these events.

Accessing shared objects

Multiple tasks can share the same data by using shared objects. A shared object is an object that regulates concurrent access by multiple tasks. You create a shared object by defining the class as Shared in the Class Workshop, and by setting the object's IsShared attribute to TRUE (explicitly or by setting the default value for the class to TRUE).

TOOL provides a shared object with the locking mechanism necessary to prevent conflicts when multiple tasks try to access or change the object's state. If one task modifies a shared object's attribute, TOOL locks the object until the change is complete. If one task invokes a method on a shared object, TOOL does not allow another task to invoke a method on the object until the first task completes its method. Other tasks attempting to invoke methods on or access/modify attributes of the same object are “blocked.” Once the first task completes the method, another task is allowed to continue.

Multiple tasks should not operate concurrently on non-shared objects (objects whose IsShared attribute is not set to TRUE). If two tasks do operate on and access the same non-shared object, the results are unpredictable.

See the *Forte 4GL Programming Guide* for more information on shared objects.

Tasks and transactions

Normally if a task is in a transaction when it starts a new task, the new task is not part of the transaction. Any transactions that begin within the new task are independent of the calling task. You can use the **transaction** clause to specify that the new task is a participant in the enclosing transaction (a dependent or nested transaction) or that it begins an independent transaction. For more information on the differences between these transaction types, refer to “Transaction Type” on page 96.

Invoking the Method

To start the new task, you must specify the object to be manipulated and the method to be invoked on the object. To specify the object, enter any object reference. The method can be any method defined for the object’s class as long as the method does not have parameters with derived C data types. (The **start task** statement cannot use parameters with derived C data types, such as structs or C-style arrays. See *Integrating with External Systems* for information about the C data types.) You can invoke any method; the method does not have to be designed especially for asynchronous processing.

You specify the values for the method parameters just as when you invoke a method synchronously. Note that even though the **start task** statement does not change output parameters, you must provide a legal value for each output parameter.

The following example illustrates starting a window as a concurrent task:

```
view_window : ViewPaintingWindow = new
  (AuctionManager = self.AuctionManager);
start task view_window.Display
  (painting_to_view = self.PaintingData[current_row],
  image_server = self.ImageManager,
  auction_server = self.AuctionManager,
  user_name = self.UserName);
```

Exception handler for invoked method

The exception handler for the method invoked by the **start task** statement should always handle the CancelException. This is because the CancelException exception is intended to cancel the task. Any methods invoked by the task’s starting method should either not handle the CancelException, or should handle it and then raise it again. This ensures that the task’s starting method will receive the CancelException and can return from the asynchronously invoked method.

Completion Clause

completion ignore option

The **completion ignore** option specifies that the method does not post the return or exception events.

completion event option

You can use the **completion event** option if the method was originally defined with return and exception events. When you use this option, the task automatically posts a return event when it completes successfully and an exception event when it terminates due to an exception.

Requesting the return and exception events automatically registers the events for the calling task. When the asynchronous method completes or terminates, Forte adds the appropriate event to the calling task’s event queue.

This registration is unlike the event registration for the **event** statement. In the **event** statement, the event is registered just before the **event** statement is ready to process the event. In the **start task** statement, the return and exception events are registered when the task is started, even though it can be much later on that your application is prepared to wait for those events. Therefore, only the “parent” task that executes the start task statement is registered for the completion event of the started task.

Return event

The return event is automatically posted to the “parent” task when the started task completes.

The return event uses the output and input-output parameters defined for the method. These parameters have whatever value is current when the task completes.

Note that if the parameter is a reference to an object and another data item is pointing to that object, the values in the object may be changed between the time that the task completes and the time that the **event** statement handles the return event. If this is a problem, you can use the **copy** option for the method’s parameters (see *A Guide to the Forte 4GL Workshops*).

Return parameter

If the method is defined as having a return type, the last parameter for the return event is a return value called “return.” This has the return value that is specified by the **return** statement.

Exception event

The exception event is automatically posted to the calling “parent” task when the started task is terminated due to an exception.

The exception event has two parameters, the **exception** parameter of type GenericException, which contains the exception that terminated the task, and the **errMgr** parameter, of type ErrorMgr, which contains the error manager for the task. These parameters let you find out why the task was terminated.

The following code illustrates using the **start task** statement with a **completion event** option to request the return and exception events and then using an **event** statement to respond to them.

Example: use of **completion event** option

Example of return event

Example of exception event

```
-- Notice output parameters are given, even though they
-- are received only on the return event.
begin transaction do
   start task self.theBid.StartBid(
      bidderName = self.theUserName,
      paintingForBid = ptg, bidValue = self.theBidValue,
      lastBidTime = self.theLastBidTime,
      bidInProgress = self.theBidInProgress)
    where completion = event, transaction = dependent;
  event loop
    when self.theBid.StartBid_return (
        nptg = paintingForBid, nvalue = bidValue,
        ntime = lastBidTime, nprogress = bidInProgress,
        nname = return) do
      -- Output values have updated data in nptg, nvalue
      -- The return value is in nname.
    when self.theBid.StartBid_exception
        (e = exception, em = errMgr) do
      -- Add error manager stack to this task and re-raise
      -- Type of e is ‘Object’
      task.ErrorMgr.AddError(errMgr=em);
      raise e;
    when <cancel_button>.Click do
      -- Abort the transaction. This will cancel the task.
      transaction.Abort(TRUE);
  end event;
exception
  when e: GenericException do
```

```
    ... unexpected...
  when e: AbortTransaction do
    ... expected...
end transaction;
```

See Auction example **Project:** Auction • **Class:** BidWindow • **Method:** Display

Transaction Clause

The **transaction** clause lets you specify whether the new task begins an independent transaction, is part of the calling task's transaction, or is not part of any transaction. You can specify any of the following types:

Transaction Type	Description
None	The new task does not begin a transaction and is not part of the enclosing transaction (this is the default).
Independent	The new task begins as an independent transaction.
Dependent	If the caller is in a transaction, the new task begins a transaction that is dependent on the enclosing transaction. Both transactions must succeed together, or both will fail. If the caller is not in a transaction, this transaction type is equivalent to **independent**.
Nested	If the caller is in a transaction, the new task begins a transaction that is nested in the enclosing transaction. If the nested transaction fails, the enclosing transaction may still succeed. If the nested transaction succeeds, it is not committed until the enclosing transaction is committed. If the caller is not in a transaction, this transaction type is equivalent to **independent**.

For further information on transaction types, see "Begin Transaction" on page 95.

Committing and aborting the transaction

When the **start task** statement begins a transaction, the method it invokes is equivalent to the statement block in the **begin transaction** statement. You commit the transaction by completing the task (this is equivalent to the **end transaction** clause). However, there is no equivalent to the statement block's exception handler. To abort the transaction, you must raise an exception that cannot be handled by the method and therefore terminates the task. You can then use the exception event if you need to be notified that the transaction was aborted. Another alternative is to use the Abort method on the TransactionHandle class (see the Framework Library online Help). If you raise an exception in the method that is handled by the method's exception handler, the transaction will not be aborted.

AbortException exception and task's dependent transactions

Unlike the transaction started by a **begin transaction** statement, the transaction started by a task does not have an automatic exception handler for the AbortException exception. Therefore, if Forte raises an AbortException exception and you do not explicitly handle it in the asynchronous method, Forte terminates the task.

Restrictions on nested transactions

In some circumstances you should not use nested transactions. You should not use nested transactions for TOOL SQL statements. Nor should you start a task with the **dependent** or **nested** options from a task that is itself running as a dependent transaction. In this case, this (second) task is a dependent participant in a first task's transaction. If the second task then starts a third task by using the **dependent** option, the third task is a dependent participant of the transaction begun by the *first task*, even though the third task was started by the second task.

Database sessions and multitasking

If you plan to use the same database session in multiple concurrent tasks, all the tasks must be in the same transaction. You can ensure that they are by using the **dependent** option of the **start task** statement.

While

The **while** statement loops through the specified statement block as long as the boolean expression is true.

Syntax

[*label* **:**] **while** *boolean_expression*
  do *statement_block*
[*exception_handler*]
end [**while**]**;**

Example

```
i = 1;
while i < 10 do
  ...
  i = i + 1;
end while;
```

Description

When the **while** statement is executed, Forte begins by evaluating the expression. If the expression is true, Forte executes the statement block. After completing the statement block, it returns to the beginning of the loop to test the expression again. Forte repeats this process until the expression is false.

You must use a boolean condition that will eventually be false, use an **exit** statement to exit the loop, or use the **return** statement to exit the method. If you do not, the **while** statement will loop infinitely. See "Exiting the While Loop" on page 152 for further information.

Expression

The boolean expression specifies a logical condition that has a value of TRUE or FALSE**.** It can include boolean variables, constants, attributes, and methods that return boolean values. See "Boolean Expressions" on page 56 for information on boolean expressions. Here is an example:

Example: boolean expression

```
while ((i < 10) and (j > 4)) and not (k = 3) do
  ...
end while;
```

Statement Block

The statement block can include any TOOL statements. You can use the **continue** statement to return to the first statement of the statement block and force another iteration of the loop. You can use the **exit** statement to pass control to the statement following the **end while**.

Example: **exit** and **continue** in statement block

```
while TRUE do
  ...processing...
  if self.Window.QuestionDialog(‘Continue?’) = BV_OK then
    continue;
  else
    exit;
  end if;
end while;
```

Exception Handling

The **exception** clause of the **while** statement provides an exception handler for the expression and the statement block in the **while** statement. See “Exception” on page 113 for information on using an exception handler.

Label

The label identifies the **while** statement for use with the **continue** and **exit** statements. When you are nesting control statements, you can use the **continue** and **exit** statements to transfer control to a labeled **while** statement. The label name must be unique for the statement block (label names share the same name scope as variables and other components).

Exiting the While Loop

To exit the **while** loop, you can use the **exit** statement to exit the **while** statement or use the **return** statement to end the method. If you do not use either of these statements, you must ensure that the boolean condition will become false, otherwise the **while** statement will loop infinitely. The following example illustrates this:

Example: exiting while loop

```
f : File = new;
... set up file name, etc...
f.Open(SP_AM_READ);
t : TextData = new;
while f.ReadLine(t,TRUE) >= 0 do
... process data in t...
end while;
```

If you are using a boolean condition to loop through a range of numbers, be sure to increment or decrement your counter.

Example: using a counter

```
x = array of TextData = new;
... fill in x ...
i : integer = 1;
while i <= x.Items do
...process x[i]...
  i = i + 1;
end while;
```

Chapter 4

Project Definition Statements

This chapter describes the statements you use to define a project and its components when writing a .pex or .cex file that you will import using the Forte workshops or the Fscript utility.

If you use the Forte workshops to define project components, Forte automatically generates these statements for you; you do not use these statements in the workshops. However, if you then export a project, you can see the statements that Forte generated if you view the resulting file. You only use these statements when you create or edit files that you will import.

To import the file into your development repository, you can use the **Compile** or **Import** command in Fscript or the **Import** command in the Project Workshop.

Begin c, dce, obb

The **begin c, dce,** or **obb** statement defines a C, DCE, or ObjectBroker project.

Syntax

begin {**c** | **dce** | **obb**} *project_name***;**
   definition_list
   [**has property** {*project_property***;**}...]
   definition_list
end [*project_name*]**;**

project_property is:

compatibilitylevel = *integer_constant*
restricted = {**TRUE** | **FALSE**}
multithreaded = {**TRUE** | **FALSE**}
libraryname = *string_constant*
extended = *name* = *string_constant* [*name* = *string_constant*]...

Description

You can use the **begin** statement to define a C, DCE, or ObjectBroker project. For complete information on this statement, and on creating and using C, DCE, and OBB projects in Forte, see *Integrating with External Systems*.

Begin class

The **begin class** statement surrounds one or more TOOL class definitions.

Syntax

begin class;
  definition_list
end [**class**];

Example

```
begin CLASS;

class ADefaultClass inherits from Framework.Object

has public  method Init;

has property
      shared=(allow=off, override=on);
      transactional=(allow=off, override=on);
      monitored=(allow=off, override=on);
      distributed=(allow=off, override=on);

end class;

method ADefaultClass.Init
  begin
    super.Init;
  end method;

end CLASS;
```

Description

The **begin class** statement allows you to define one or more TOOL classes in a .cex file. It provides the same support for forward referencing as the **begin tool** statement and is the first statement in the .cex file that is created when you export a class. You can have only one **begin class** statement per file.

To import a class definition from the file into your current project, you can use the **Compile** command in Fscript or the **Import Class** command in the Project Workshop. Forte imports only the classes you specify into the current project.

If there is more than one definition for the same class or component name, Forte uses the last definition.

Definition List

The definition list for the **begin class** statement can include any number of **class**, **interface**, **constant**, **cursor**, and **service** statements to define components for the current project.

Forward registration of class names

Note that TOOL allows you to include more than one definition for the same class, interface, constant, cursor, or service object within the definition list. The last definition in the list is the definition that Forte uses for the project. The advantage of being able to include more than one definition for the same component is that you can forward register class names in the definition list. This way, one class definition can reference another class before that class has been completely defined.

Begin tool

The **begin tool** statement defines a TOOL project.

Syntax

begin [**tool**] *project_name***;**
  [**includes** *project_name***;**]...
  [**has property restricted** = {**TRUE**|**FALSE**}**;**]
  definition_list
  [**has property** {*project_property***;**}...]
end [*project_name*]**;**

project_property is:

startingmethod = (**class** = *class_name*, **method** = *method_name*)
compatibilitylevel = *integer_constant*
libraryname = *library_name*

Description

The **begin tool** statement allows you to define a TOOL project in a .pex file. To import the project definition from the file into your development repository, you can use the **ImportPlan** command in Fscript or the **Import** command in the Repository Workshop.

You can have more than one **begin tool** statement for the same project. These statements can be in the same file or in different files. If the project already exists, Forte simply adds the new definitions to the existing project.

If there is more than one definition for the same project component (for example, more than one definition for the same class name), Forte uses the last definition.

Project Name

The project name can be any legal Forte name. If the name is unique, Forte creates a new project. You can use an existing project name, which will add new definitions to the existing project, but the existing name must be for a TOOL project (not an external project). When you specify an existing TOOL project name, Forte adds the definitions in the **begin tool** statement to the existing project.

Includes Clause

The **includes** clause specifies a supplier plan for the project you are defining. A supplier plan can be any project or library. If your project needs to access definitions or services defined in another project or in a library, you must include that supplier plan as part of your project definition. Repeat the **includes** clause any number of times to specify each of the supplier plans for the project.

Definition List

The definition list for a project can include any number of **class**, **constant**, **cursor**, **interface**, and **service** statements (described in this chapter).

Forward registration of class names

Note that TOOL allows you to include more than one definition for the same class, constant, cursor, interface, or service object within the definition list. The last definition in the list is the definition that Forte uses for the project. The advantage of being able to include more than one definition for the same component is that you can forward register class names in the definition list. This way one class definition can reference another class before it has been completely defined.

Has Property Clause

The **has property** clause lets you specify the project properties. These are the same properties you can set for a project in the Project Workshop.

You can have more than one **has property** clause within the **begin tool** statement. If you set the same property more than once, Forte uses the last setting.

startingmethod option

The **startingmethod** option lets you specify the startup class and method for the project. The **class** clause specifies the class of the startup object for the application. You can specify any class defined in the project. The **method** clause specifies the method that will be invoked on the startup object. You can specify any method defined for the class.

compatibilitylevel option

The **compatibilitylevel** option lets you specify the compatibility level for the project. Normally, if you plan to deploy a new release of your application, you should raise its compatibility level. This lets you install and run the new release of the application in the same environments where older releases of the application are installed. You must raise the compatibility level of the project if you make any changes to the project except the following:

- change method source code (but not method parameters)
- add new classes

restricted option

A project is defined as having restricted availability because it can run only on particular hardware or software. If your project includes any restricted projects (either external or TOOL) or libraries, you can use the service objects provided by those plans but you cannot create objects using their classes. This is because the necessary hardware or software required by the restricted plan may not be available.

However, under some unusual circumstances, your project may need to run the restricted plan’s code in the client partition (that is, it needs to create objects from its classes). In this case, you can declare your TOOL project as also being restricted by setting the **restricted** option to TRUE. If you declare your project as restricted, Forte allows you to use the classes of all the restricted supplier plans. However, this has serious repercussions on how you can partition the project.

Note that you must set the **restricted** property before the definition list. This is true only for the **restricted** property. Other properties can be set before and after the definition list.

library name option

A **library name** is used when the project is configured as a library or included within a library distribution. When there is more than one library within a library configuration, all the library names must be unique. The library name can be any length, however, on platforms where there is an eight-character limit for file names, the library name will be truncated to eight characters.

Class

The **class** statement creates a custom class.

Syntax

class *class_name* [**is mapped**] **inherits** [**from**] *superclass*
  [**implements** *interface,* [*interface*]...]
[[**has public | has private**] *component_definitions*]...
[**has property** [*property*]...
[**has file** *filename*;]
end [**class**]**;**

property is one of:

shared= (**allow** = {**on**|**off**} [, **override** = {**on**|**off**}]
  [, **default** = {**on**|**off**}]);

transactional= (**allow** = {**on**|**off**} [, **override** = {**on**|**off**}]
  [, **default** = {**on**|**off**}]);

monitored = (**allow** = {**on**|**off**} [, **override** = {**on**|**off**}]
  [, **default** = {**on**|**off**}]);

distributed= (**allow** = {**on**|**off**} [, **override** = {**on**|**off**}]
  [, **default** = {**on**|**off**}]);

restricted = (**TRUE**|**FALSE**)**;**

extended = (*name*=*value*[,*name*=*value*]...);

Example

```
class Artist inherits from Object
has public
  -- Attributes
  Name : string;
  Country : string;
  Born : integer;
  Died : integer;
  School : string;
  Comments : TextData;
  -- Methods
  method FillInData(ioObject : BasicIO);
  method WriteToLog();
  method Init();
  -- Properties
  has property
    shared = (allow = off);
    transactional = (allow = on, default = on);
end class;
```

Description

Typically, you create classes using the Class Workshop. However, the **class** statement allows you to define custom classes in a file. To import the class definition from the file into your development repository, you can use the **Compile** command in Fscript or the **Import** command in the Project Workshop.

Class Name

The class name provides a name for the new class. If a class by that name already exists, your new definition replaces the existing definition.

We recommend that you adopt a naming convention to distinguish your custom classes from the Forte system classes.

Implements Clause

The implements clause specifies one or more interfaces that the new class will implement. The class can implement any interface that was previously defined or forward registered.

To implement an interface, the class must define all methods, events, and event handlers included in the interface. If the class does not define all these components, you will get a compile error.

Is Mapped Clause

Use for window classes

If the class you are defining is a window, that is, a subclass of the UserWindow class, you must include the **is mapped** clause in the **class** statement. This indicates that the class is a user window class, and that the **has file** clause (described below) specifies the window definition file. Forte compiles both the class definition and window definition file, mapping the class attributes to the corresponding window widgets by name.

Inherits Clause

Specifying the superclass

The **inherits** clause specifies one superclass from which the new class will inherit attributes, methods, events, event handlers, and constants. This can be any custom class or any system class identified as “Superclass for Custom Class” in the documentation. This is required.

Public and Private Definitions

The **has public** clause defines public attributes, methods, events, event handlers, and constants for the class. “Defining Class Components” describes how to define these individual components. You can use this clause any number of times within the **class** statement and in any combination with the **has private** clause. If the **has public** clause is the first clause in the class definition, the **has public** key words are optional. Otherwise, you must include the key words with the component definitions.

The **has private** clause defines private attributes, methods, event handlers, and events for the class. “Component Definitions” on page 161 describes how to define these individual components. You can use this clause any number of times within the **class** statement and in any combination with the **has public** clause.

Has File Clause

Window definition file

If the class you are defining is a window, that is, a subclass of the UserWindow class, you must use the **has file** clause to specify the window definition file for the class. This is required if you use the **is mapped** clause. The window definition file must be a file that you created using the Forte Window Workshop. To specify the file name in the **has file** clause, you must enclose the file name in single quotes.

Mapping widgets to class attributes

In order for the widgets in the window definition to map to the data attributes in the class, they must have the same name and a compatible data type (see *A Guide to the Forte 4GL Workshops* for information on mapping data types). If there is no matching name, the widget will be included in the class but you will not be able to display data in it. If the type is incompatible, you will get an error.

Component Definitions

The components you can define for a class include:

- attributes
- virtual attributes
- events
- methods
- event handlers
- constants

Attributes

To define an attribute, you must specify the name and type. The syntax is:

[**attribute**] *attribute_name* **:** *type*
  [**has property** [**extended** = (*name*=*value*[,*name*=*value*]...)]];

The attribute name is any legal Forte name. It must be unique for the class. The attribute type can be any simple type or any class.

The following example shows attribute definitions:

```
PaintingForBid : Painting;
BidValue : DecimalData;
```

Virtual Attributes

To define a virtual attribute, you must specify the attribute name, the attribute type, and two expressions that determine the value of the attribute when it accessed or assigned. The syntax is:

virtual [**attribute**] *attribute_name* **:** *type* = (**get =** *expression***,**
  [**set** = *expression*])
  [**has property** [**extended** = (*name*=*value*[,*name*=*value*]...)]];

The attribute name is any legal Forte name. It must be unique for the class. The attribute type is any simple data type or any class.

Get expression

The **get** parameter provides an expression to be executed when the program accesses the value of the attribute. The value of the expression is the value of the virtual attribute. This can be any expression with a data type that is compatible with the attribute's data type.

For a private virtual attribute, the expression can reference any components defined in the current class and its superclasses, or any global components. For a public virtual attribute, the expression can reference only public attributes, methods, event handlers, events, and constants.

Since a class definition cannot point to an object, Forte assumes that all references to attributes, methods, and so on are for the current object.

The following example illustrates using the return value of a method as the value of the virtual attribute:

Example: virtual attribute

```
class weather inherits from Object
has public
  -- Method sets lowf attribute with input in centigrade
  method setctemp (temp_in_centigrade : integer);
  -- Attribute lowf is temperature in fahrenheit
  lowf : integer;
  -- Virtual attribute shows temp in centigrade.
  virtual lowc : integer =
      (get = (5.0/9.0) * (lowf - 32),
      set = SetCtemp(lowc));
end class;
method weather.setctemp(temp_in_centigrade : integer)
begin
  self.lowf = (9.0/5.0) * temp_in_centigrade + 32;
end;
```

In the **get** expression, you can use the virtual attribute name as an input-output or output parameter for a method. Forte uses this "return" value as the value for the virtual attribute. Any other use of the virtual attribute name is illegal.

Set expression

The **set** parameter provides an expression to be evaluated when the program assigns a value to the attribute. This expression usually invokes a method to update some data. This expression is optional. If you do not specify it, the virtual attribute will be read only.

For a private virtual attribute, the expression can reference any components defined in the current class and its superclasses, or any global components. For a public virtual attribute, the expression can reference only public attributes, methods, event handlers, events, and constants. Forte assumes that all references to attributes, methods, and so on are for the current object.

In the **set** expression, you can use the virtual attribute name to represent the value that the user assigned to the attribute. Typically, the expression contains a method that uses the virtual attribute name as one of its parameters. (Note that you cannot update the value of an attribute by setting its value directly to the value of the virtual attribute; instead, you must invoke a method to update it.)

If the **set** expression produces a value, such as a return value from a method, Forte ignores it.

Events

You define an event by specifying an event name and an optional list of parameters. The syntax is:

event *event_name* [(*name* **:** *type* [= *value*] [**,** *name* **:** *type* [= *value*]]...)]
 [**has property** [**extended** = (*name*=*value*[,*name*=*value*]...)]];

The event name is the name you will use to identify the event in the **post** and **event** statements. This can be any legal Forte name. It must be unique for the class.

The parameter list defines one or more parameters for the event. When you use the **post** statement to generate the event, you can pass one value for each parameter that you define here. For each parameter, you must specify a name and a data type. The data type can be any simple type or any class.The optional value for the parameter specifies a default value to be used when the event is posted without specifying a value for the parameter.

Example: event definition

```
event BidCompleted (newBid : DecimalData,
  timeOfBid : DateTimeData,
  whoBid : TextData);
event BidCancelled;
```

No derived C data types

Note that event parameters cannot have derived C data types, such as structs or C-style arrays. See *Integrating with External Systems* for information about the C data types.

Methods

To include a method in a class definition, you must specify the method name, the method parameters, and the method return type. If you want to create return and exception events to be used with the **start task** statement, you must specify names for these events. You provide the code for method body separately, using the **method** statement. The syntax is:

[**method**] *method_ name* [(*parameter_list*)] [**:** [**copy**] *return_type*]
  [**where completion** = (*event_setting* [, *event_setting*])
  [**has property** [**extended** = (*name*=*value*[,*name*=*value*]...)]];

parameter _list is:

([**copy**] [**input** | **output** | **input output**] *name* **:** *type* [= *value*]
  [**,** [**copy**] [**input** | **output** | **input output**] *name* **:** *type* [= *value*]]...)

event_setting is:

return = *event_name* | **exception** = *event_name*

When you use the **method** statement to write the body of the method, you must again specify the method name, parameters, and return type for the method. These must match exactly with those you specify in the class definition, including the type and default values.

Use of **method** key word

Note that the **method** key word is only optional when the method has a parameter list. If the method does not have a parameter list, you must include the **method** key word.

Method name

The method name specifies the name that you must use to invoke the method. This name must be unique for the class unless you wish to override or overload the method.

If you use a unique name, Forte creates a new method for the class with a new method name.

Overriding

If you use the name of an inherited method name and specify the same parameters used in that method (see below), Forte "overrides" the inherited method for the current class. This allows you to create a different variation of the method that will be invoked with the same method name. When you invoke a method on the object, Forte uses the method you defined specifically for the class rather than the inherited method. (You must provide code for the new method using the **method** statement for this class.)

Overloading

If you use the name of an existing method for the class but you specify different data types for the parameters (see below), Forte "overloads" the method. This allows you to create two similar method with the same name. In this case, when the method is invoked for the object, Forte uses the parameter data types as well as the method name to determine which method to use. (You must provide code for overloaded method using the **method** statement.) See *A Guide to the Forte 4GL Workshops* for information about overriding and overloading methods.

The following example defines a method with a method name of WriteToLog with no parameters and no return type.

```
method Artist.WriteToLog()
```

Method parameters

The parameter list defines one or more parameters for the method. For each parameter, you must specify a name and data type. The data type can be any simple data type or any class. The following example illustrates:

```
method DeleteBidForPainting (name : TextData) : boolean;
```

If you want to "overload" the method (described above), be sure to specify different parameter data types than those for the method with the same name. When you invoke a method and the object has more than one method with the same name, Forte uses the parameter data types (not the parameter names) to distinguish between them.

Restrictions for overloading

There are two restrictions for overloading methods. First, you cannot define two methods where the only difference is that one has a portable integer parameter and the other has a non-portable integer parameter. Second, you cannot define two methods where the only difference is that one has a boolean parameter and the other has any unsigned integer parameter. Both of these conditions will cause compiler errors. See "Integer Data Types" on page 59 for information on portable and non-portable integers.

input, **output**, and **input output** options

The default for a method parameter is input only. You can use the **output** or **input output** options to specify that the parameter is for output only or for both input and output.

Example: input and output parameters

```
method StartBid(input bidderName : TextData,
  output paintingForBid : Painting,
  output bidValue : DecimalData,
  output lastBidTime : DateTimeData,
  output bidInProgress : boolean) : TextData;
```

Class parameters

When a parameter has a class for its type, Forte passes a *reference* to the object, not the object itself. This means that even if the parameter is for input only, if the method makes changes to the object, these changes are reflected when you return from the method. This is because both the invoking method and the invoked method are referencing the same object.

copy option

The **copy** option allows you to prevent the invoking method and invoked method from referencing the same object. When you specify **copy** for a class parameter, Forte makes a copy of the object and passes a reference to that copy as the parameter. For an output parameter, Forte returns a reference to the copy. The **copy** option is also useful for improving the efficiency of distributed applications. If the invoking method and the invoked method are going to be on separate partitions, using a copy of the object rather than sharing a single object may reduce communication costs. In most cases, Forte will try to use the most efficient communication that is possible.

Default value

Input and copy-input parameters have an optional default value. The optional value for the parameter specifies the default value to be used when the programmer invokes the method without assigning a value to the parameter. Providing a default value for the parameter makes the parameter optional (the programmer does not have to assign a value to it when invoking the method). Parameters that do not have default values are required (the programmer must assign a value to it when invoking the method).

For simple parameters, the default value can be any constant that is compatible with the data type of the parameter. For class parameters, the default value must be the NIL constant.

Note that you can use a named constant as a default value, but only the current value of the named constant at that point in time is used. If the named constant later changes its value, the parameter's default value is not changed unless the code gets recompiled.

Method return type

The method's return type specifies the data type of the value that you can pass back from the method with the **return** statement. The return value can be any simple data type or any class. If you do not specify a return type, you cannot use the **return** statement to pass a value back to the invoking method.

Example: return types

```
method FindBidForPainting (name : TextData) : Bid;
method GetPaintingList () : Array of Painting;
method OpenNewList();
```

The optional **copy** option for the method's return type lets you request a reference to a copy of the object, rather than a reference to the original object. This is exactly like the **copy** option for parameters (described above).

where completion option

If you are planning to invoke the method asynchronously using the **start task** statement, you may wish to request an event to be posted when the task completes successfully or is terminated due to an unhandled exception. In order to request these events to be posted, you must first define them for the method. Note that you cannot specify completion events on a method that is overriding another method. This is because the overriding method inherits the completion events from the original method.

return option

The **return** option declares a return event with the name you specify. Forte posts this event when the task completes successfully (if you used the **completion event** option in the **start task** statement to request the event). The return event has the output and input-output parameters defined for the method. These parameters have whatever value is current when the task completes. In addition, if the method is defined as having a return type, the last parameter for the return event is a return value called “return.” This has the return value that is specified by the **return** statement.

exception option

The **exception** option declares an exception event with the name you specify. Forte posts this event when the task is terminated due to an unhandled exception (if you used the **completion event** option in the **start task** statement to request the event). This event has two parameters, the **exception** parameter of type Object, which contains the exception that terminated the task, and the **errMgr** parameter of type ErrorMgr, which contains the error manager for the task.

See “Completion Clause” on page 148 for more information about completion events.

Event Handlers

To include a named event handler in a class definition, you must specify the event handler name and the event handler parameters. You provide the code for event handler separately, using the **event handler** statement. The syntax is:

event handler *handler_ name* [(*parameter_list*)]
  [**has property** [**extended** = (*name*=*value*[,*name*=*value*]...)]];

parameter _list is:

([**copy**] [**input**] *name* **:** *type* [= *value*]
  [**,** [**copy**] [**input**] *name* **:** *type* [= *value*]]...)

Note that when you use the **event handler** statement to write the event handler, you must again specify the handler name and parameters. These must match exactly with those you specify in the class definition.

Handler name

The handler name specifies the name that you must use to identify the event handler. This name must be unique for the class unless you wish to override an inherited event handler.

If you use a unique name, Forte creates a new event handler for the class with a new event handler name.

Overriding

If you use the name of an inherited event handler, Forte “overrides” the inherited event handler for the current class. This allows you to create a different variation of the event handler that can be registered using the same event handler name. When you register an event handler for the current class, Forte uses the event handler you defined specifically for the class rather than the inherited event handler. (You must provide code for the new variation of the event handler using the **event handler** statement for this class.)

No overloading

Forte does not allow you to overload event handlers. There can only be one event handler with a given name in the class.

Handler parameters

The parameter list defines one or more parameters for the event handler. When you register the event handler with the **register** statement, you can pass one value to the event handler for each parameter that you define here. For each parameter, you must specify a name and data type. The data type can be any simple data type, any class, or any array type. The following example illustrates:

```
event handler ArtObjectWindow.ArtObjectHandler
  (input artType: string)
```

input option

Event handler parameters are always for input only. If desired, you can use the **input** key word in your event handler definition for clarity. See *A Guide to the Forte 4GL Workshops* for information about parameters for event handlers.

Class parameters

When a parameter has a class for its type, Forte passes a *reference* to the object, not the object itself. This means that even though an event handler parameter is for input only, if the event handler makes changes to the object, these changes are reflected outside the event handler. This is because both the method that registers the event handler and the event handler itself are referencing the same object.

copy option

The **copy** option allows you to prevent the method and the event handler from referencing the same object. When you specify **copy** for a class parameter, Forte makes a copy of the object and passes a reference to that copy as the method handler parameter.

Default value

Event handler parameters have an optional default value. The optional value for the parameter specifies the default value to be used if the programmer registers the event handler without assigning a value to the parameter. Providing a default value for the parameter makes the parameter optional (the programmer does not have to assign a value to it when registering the event handler). Parameters that do not have default values are required (the programmer must assign a value to it when registering the event handler).

For simple parameters, the default value can be any constant that is compatible with the data type of the parameter. For parameters whose type is a class, the default value must be the NIL constant.

Note that you can use a named constant as a default value, but only the current value of the named constant at that point in time is used. If the named constant later changes its value, the parameter's default value is not changed unless the class gets re-imported.

Constants

To define a class-level constant, use the **constant** statement within the definition list. See "Constant" on page 171 for information.

Has Property Clause

The **has property** clause allows you to specify whether objects of the specified class can be distributed, shared, transactional and/or monitored. These class properties add certain runtime behavior to objects that are instances of the class. Briefly the properties are:

Property	Description
Distributed	Allows an object of this class to be sent to a remote partition.
Shared	Lets you create shared objects, which allow multiple tasks to access and reliably change the object's data concurrently.
Transactional	Lets you create transactional objects which can participate in a transaction.
Monitored	Indicates that objects of this class may be displayed and mapped to a window widget.

Each of these properties is associated with an attribute of the Object class. The following table describes the Object attribute for each class property:

Class Property	Object Attribute	Description
Distributed	IsAnchored	Setting IsAnchored to TRUE anchors the object to the partition in which it was created. An attempt to send such an object to a remote partition confirms that the object's class definition included the Distributed property.
Shared	IsShared	Setting IsShared to TRUE uses locking to synchronize access from multiple concurrent tasks. By default IsShared also sets IsAnchored.
Transactional	IsTransactional	Setting IsTransactional to TRUE marks the object as transactional. TOOL logs such objects before they're updated in a transaction.
Monitored	none	No associated Object attribute.

See the Framework Library online Help for information on the IsAnchored, IsShared and IsTransactional attributes of the Object class.

allow parameter

Setting the **allow** parameter to ON indicates that objects of the class may set the associated Object attribute to TRUE. If you set **allow** to OFF, then the associated attribute cannot be set to TRUE.

override parameter

The **override** parameter determines if subclasses inherit the settings for the **allow** and **default** parameters and specifies whether the subclass can change the settings of these parameters.

The ON setting specifies that subclasses do not inherit the settings, but can change their values. The subclass is initially set to the default values for the property, but can then set the value as appropriate. The default value for the override parameter is ON.

The OFF setting specifies that subclasses inherit their superclass' values and the subclass cannot change them. If you change a superclass with existing subclasses from override ON to override OFF, all its subclasses will be changed to the use the same values as the superclass **default** parameter.

The **default** parameter provides an initial setting of the associated Object attribute. The **default** parameter can only be set to ON if **allow** is also set to ON. Normally, the initial setting for the associated attribute is FALSE. To change this initial value, set the **default** parameter to ON.

In the following example, all object instances of the Artist class will automatically be transactional, but only those explicitly set will be shared:

Example: default parameters

```
class Artist inherits from Object
   ...
   has property
    shared = (allow = on); -- no default
    transactional = (allow = on, default = on);
end class;

 a1 : Artist = new; -- Transactional (default)
a2 : Artist = new(IsShared = TRUE); -- Transactional and shared
```

The **default** parameter is not applied to objects created with the Clone method. You can safeguard against this by setting the associated attribute in an Init method.

Default settings and performance considerations

When you create a class, it is given default settings values for each of the class properties. The default property settings are briefly described in the table below:

Class Property	Default Values	Description

Distributed	allow = off
	override = on
	default = off
		The object cannot be sent to a remote partition.

Shared	allow = off
	override = off
	default = off
		Access to the object's attributes and methods is not regulated in a multitasking environment.

Transactional	allow = off
	override = off
	default = off
		The object will not participate in a transaction.

Monitored	allow = on
	override = on
	default = off
		The system may map the object's values to a window widget.

In order to improve the runtime performance of a class, in both a client and server, all properties should be explicitly turned off. For example:

```
distributed = (allow = off, override = off);
shared = (allow = off, override = off);
transactional = (allow = off, override = off);
monitored = (allow = off, override = off);
```

Whenever possible, you should turn off as many of the properties as you can without impacting the runtime behavior of your application. See the *Forte 4GL Programming Guide* for more information on distributed, shared, transactional and monitored objects.

Has Property Restricted Clause

If the current project has a supplier project or library that is restricted and the class you are creating in the current project references components in the restricted plan, you can declare the class as "restricted." When the class itself is restricted, this allows you to create objects from the class, not just service objects.v

Has Property Extended Clause

The **has property extended** clause for a class allows you to set any number of extended properties for a class or any of its elements. Extended properties are simply user-defined name-value pairs.You can assign arbitrary name-value pairs to the class or its components for any purpose. For example, you can use extended properties for comments.

Example Class Definition

The following example illustrates a complete class definition.

```
class Bid inherits from Object
has private
  PaintingForBid : Painting;
  BidValue : DecimalData;
  LastBidTime : DateTimeData;
  LastBidder : TextData;
  BidInProgress : boolean;
has public
  method Init();
  method CompleteBid (bid : DecimalData);
  method GetValues (
    output paintingForBid : Painting,
    output bidValue : DecimalData,
    output lastBidTime : DateTimeData,
    output bidInProgress : boolean);
  method SetValues (
    paintingForBid : Painting = NIL,
    bidValue : DecimalData = NIL,
    lastBidTime : DateTimeData = NIL,
    lastBidder : TextData = NIL,
    bidInProgress : boolean = FALSE);
  method StartBid(
    input bidderName : TextData,
    output paintingForBid : Painting,
    output bidValue : DecimalData,
    output lastBidTime : DateTimeData,
    output bidInProgress : boolean) : TextData
  where completion = (return = StartBid_return,
        exception = StartBid_exception);
  ...more methods...
  event BidCompleted (newBid : DecimalData,
    timeOfBid : DateTimeData, whoBid : TextData);
  event BidCancelled;
  event BidStarted (who : TextData);
has property
  transactional = (allow = on, default = on);
  shared = (allow = on, default = on);
  distributed = (allow = on);
end class;
```

See AuctionServerProject example

Project: AuctionServerProject • **Class:** BidWindow • **Method:** Display

Constant

The **constant** statement declares a named constant.

Syntax

constant *name* = *value*;

Example

```
constant seconds_per_hour = 3600;
constant PI = 3.14159268;
constant CompName = ‘Forte Software Inc.’;
```

Description

The **constant** statement allows you to define a constant in a file using the Forte Fscript utility. You can use the **constant** statement to declare a named constant as part of a project definition or as part of a class definition.

If you declare a constant as a project component, its scope is the entire project. Any other component in the project can reference the constant. See “Begin tool” on page 157 for information about defining projects and their components.

If you declare a constant as a class component, its scope is limited to that class. In order to reference the constant from outside the current class, you must use a qualified name. See “Qualified Names” on page 53 for information about qualified names. See “Class” on page 159 for information about defining classes and their components.

Constant Name

The constant name can be any legal Forte name and must be unique for the current statement block. Because constants share the same name scope as several other components, if the constant has the same name as a component in an enclosing scope, the new named constant will “hide” the existing component. See “Name Resolution” on page 52 for information on name resolution.

Constant Value

The constant value can be any numeric or string value. The data type of the value determines the data type of the constant. See Chapter 2, “Language Elements,” for information about how to specify a numeric or string constant.

Once you specify the value for a constant, you can change it only by recompiling the **constant** statement, or through the Project Workshop (for project constants) or the Class Workshop (for class constants).

Cursor

The **cursor** statement declares a database cursor.

Syntax

cursor *name* [(*name***:** *type* [= *value*] [**,** *name* **:** *type* [= *value*]]...)]
begin
select_statement
[**for** {**read only**| **update** [of *column* [**,** *column*]...]}];
end;

Example

```
cursor artist_cursor (name : string)
begin
  select name, country, born, died, school, comments
  from artist_table
  where name LIKE :name;
end;
```

Description

The **cursor** statement allows you to define cursors in a file using the Forte Fscript utility. You can use the **cursor** statement to declare a cursor for use with the project. The example above illustrates this.

Cursor Name

The cursor name can be any legal Forte name and must be unique for the project.

Placeholders

When you declare the cursor, you have the option of declaring one or more placeholders. These are names that represent values that will be supplied at runtime. After you declare placeholders, you use them in the **where** and **having** clauses of the **select** statement in the cursor definition. Then, when you open the cursor, you set the values of the placeholders as part of the **open cursor** statement.

Name and type

Each placeholder has a name, a type, and an optional default value. The name can be any legal Forte name. The type can be any Forte simple data type or any DataValue subclass.

Default values

The optional default value is used when you do not specify the value for the placeholder in the **open cursor** statement. This can be any value that is compatible with the data type of the placeholder.

Cursor Select Statement

The **select** statement for the cursor selects the database rows for processing. Forte executes this **select** statement when you use the **open cursor** statement to open the cursor. You can use the **fetch cursor** statement to move the cursor through the result set.

The syntax for the **select** statement associated with a cursor is:

select [**all** | **distinct**] (* | *column_list*) **from** *table_name* [**,** *table_name*]...
  [**where** *search_expression*]
  [**order by** *column* [**asc** | **desc**] [, *column* [**asc** | **desc**]]...]
  [**group by** *column_name* [**,** *column_name*]...]
  [**having** *search_expression*]**;**

See *Accessing Databases* for descriptions of the individual clauses for the **SQL select** statement.

Using placeholders

In the **where** and **having** clauses, you can use the placeholders you declared in the placeholder list. Because the placeholders are Forte names, you must preface them with colons to distinguish them from column names.

For Clause

By default, a cursor can be used for reading only. The **for update** clause lets you allow updating as well as reading. You can either allow updating for all columns or limit updating to a specified list of columns.

Read only cursor

The **read only** option limits the cursor to reading only, which is the default. When you use the **sql fetch** statement with a read only cursor, you can access the values in the row but you cannot update or delete it. Using a read only cursor keeps the data available to others while you are working with the rows. Note that your particular DBMS may not support read only cursors.

Update cursor

The **update** option allows the cursor to be used for updating. This provides a lock on the data to prevent inconsistencies during the update. When you open a cursor that has been declared for updating, other users will not be able to access the result set (and possibly other data) until you close the cursor. Although it is not required, we recommend that you use the **of** clause to specify the particular columns that can be updated. This allows Forte to optimize the code and prevents updating of columns that should not be changed. Any columns that you do not include in the **of** clause are available for reading only.

Event Handler

The **event handler** statement defines an event handler for a class.

Syntax

event handler *class***.***handler_name* [(*parameter_list*)]
begin
   [**preregister** *statement_list*]...
   [[**postregister**] *statement_list*]
   [**when** *event_specification* **do** *statement_block*]...
   [*exception_handler*]
end [**event**]**;**

Example

```
event handler ArtObjectWindow.ArtObjectHandler
  (input artType: string)
begin
  preregister
    -- Use the parameter passed to the handler to set the
    -- initial value of the object_type widget’s mapped
    -- attribute.
    artobject_data.object_type.SetValue(artType);

  -- Validate the object_type field.
  when <artobject_data.object_type>.AfterValueChange do
    retval : TextData = self.artobject_data.ValidateType();
    if retval != NIL then
      -- Error in validation.  Print message and get rid
      -- of pending events.
      self.Window.MessageDialog(messageText = retval);
      self.Window.PurgeEvents();
    end if;
  -- Validate the whole thing.
  when <artobject_data>.AfterValueChange do
    retval : TextData = self.artobject_data.ValidateAll();
    if retval != NIL then
      -- Error in validation.  Print message and get rid
      -- of pending events.
      self.Window.MessageDialog(messageText = retval);
      self.Window.PurgeEvents();
    end if;
end event;
```

Description

The **event handler** statement provides the source code for the event handler you create in a **class** statement. This is the code that provides the event handler's functionality.

When you use the **event handler** statement, you must specify the handler name and parameters. These must match the event handler name and parameters you specified for the event handler in the **class** statement.

Class and Handler Name

You must specify the class to which the event handler belongs. Because you can override inherited event handlers, Forte uses the class name to assign the event handler to the class.

The handler name is the name you must use to identify the event handler. This must be the same as the name you specified for the event handler in the class definition.

Note that you can override an event handler, but you cannot overload it. There can only be one event handler with a given name in the class.

Parameters

The handler parameters provide the mechanism for passing values to the event handler. This clause must match exactly with the handler's parameter list you specified in the **class** statement. The syntax is:

([**copy**] [**input**] *name* **:** *type* [= *value*]
  [, [**copy**] [**input**] *name* **:** *type* [= *value*]]...)

See "Event Handlers" on page 166 for information about the handler's parameter list.

Preregister Clause

The optional **preregister** clause provides a list of statements to be executed *before* Forte registers the events in the **when** clause list. You can use any TOOL statements in the **preregister** clause except the **return** and **exit** statements.

The **preregister** clause is especially useful for including other named event handlers in the current event handler. Using the **register** statement in the **preregister** clause is the primary mechanism for including one event handler's code within another event handler. For example:

```
-- Registering handlers from both self and super in an event loop.
event loop
    preregister
       register self.ArtistHandler();
       register self.ResetHandler();
       -- Use the inherited event handler to handle exit button
events.
       register super.ExitHandler();
end event;
```

See InheritedWindow example

Project: InheritedWindow • **Class:** ArtistDataEntryWindow • **Method:** Display

See "Register" on page 131 for further information about using the **register** statement.

Forte registers the events in the **preregister** clause before registering the events in the **when** clauses of the event handler. Therefore, if the same event is registered by the **preregister** clause and a **when** clause, the **when** clause will supercede the **preregister** clause. The event handling code in the **preregister** clause will become inactive.

The **preregister** clause is also useful for creating the objects that will be posting the events you are registering for. For example, you can include the **start task** statement that will create the TaskDesc object on which return events will be posted. The following example illustrates this:

```
event handler ArtSeller.ArtSellerHandler()
begin
  preregister
    start task self.TheSale.StartSale()
      where completion = event;
  -- When the sale has started, display a message.
  when self.The Sale.StartSale_return do
    self.Window.MessageDialog( messageText = ‘Sale has started.’);
end event;
```

Scoping for variables

The statement list for the **preregister** clause determines the scope for any variables or constants that are declared within it. Variables and constants that you declare within the statement list are available within the **preregister** clause only and cannot be accessed by the rest of the event handler.

Exception handling

The statement list in the **preregister** clause does not provide an exception handler. If exceptions are raised within the **preregister** clause, they are handled by the **event** statement that registers the event handler. See “Event” on page 106 for information about exception handling for the **event** statement.

Of course, you can include a **begin/end** statement block in the **preregister** clause, and use the exception handler within the **begin/end** statement to handle the exceptions raised within the **begin/end** block. This way, the exceptions raised within the **begin/end** block can be handled within the block, rather than by the enclosing **event** statement.

Postregister Clause

The optional **postregister** clause provides a list of statements to be executed *after* Forte registers the events in the **when** clause list, but before the events are handled by the event handler. You can use any TOOL statements in the **postregister** clause except the **return** and **exit** statements.

The **postregister** clause is useful for ensuring that the code that posts an event is always executed after the event statement has registered for that event (and so is ready to receive it). For example:

```
event handler ArtSeller.ArtSellerHandler()
begin
  postregister
    -- StartSale posts an event when it’s ready for input.
    start task self.TheSale.StartSale();
  -- When StartSale has posted the event, do something.
  when self.The Sale.ReadyforInput do
    ...start another task related to input...
end event;
```

You can use the **register** statement in the **postregister** clause to include named event handlers in the current event handler.

Forte registers the events in the **postregister** clause after registering the events in the **when** clauses of the event handler. Therefore, if the same event is registered by the **postregister** clause and a **when** clause, the **postregister** clause will supersede the **when** clause. The event handling code in the **when** clause will become inactive. See "Register" on page 131 for further information about using the **register** statement.

Scoping for variables

The statement list for the **postregister** clause determines the scope for any variables or constants that are declared within it. Variables and constants that you declare within the statement list are available within the **postregister** clause only and cannot be accessed by the rest of the event handler.

Exception handling

The statement list in the **postregister** clause does not provide an exception handler. If exceptions are raised within the **postregister** clause, they are handled by the **event** statement that registers the event handler. See "Event" on page 106 for information about exception handling for the **event** statement.

Of course, you can include a **begin/end** statement block in the **preregister** clause, and use the exception handler within the **begin/end** statement to handle the exceptions raised within the **begin/end** block. This way, the exceptions raised within the **begin/end** block can be handled within the block, rather than by the enclosing **event** statement.

postregister key word

Note that if you include both the **postregister** clause and the **preregister** clause in the same event handler, you must use the **postregister** key word.

When Clause

The **when** clause identifies the event that you wish to handle and provides the corresponding code for that particular event. First you must specify which event you wish to handle. Second you declare a series of variables to receive the parameters that will be passed with the event. Finally, you must provide the statement block to be executed when the event is posted.

Event specification

If the event is for the current object, you need only specify the event name. Otherwise, you must reference the object that will produce the event and specify the event name using dot notation. The event can be any event defined for the object's class or one of its superclasses.

Example: event specification

```
-- Validate the object_type field.
when <artobject_data.object_type>.AfterValueChange do
  retval : TextData = self.artobject_data.ValidateType();
  if retval != NIL then
    ---- Error in validation.  Print message and get rid
    -- of pending events.
    self.Window.MessageDialog(messageText = retval);
    self.Window.PurgeEvents();
  end if;
-- Validate the whole thing.
when <artobject_data>.AfterValueChange do
  retval : TextData = self.artobject_data.ValidateAll();
  if retval != NIL then
    -- Error in validation.  Print message and get rid
    -- of pending events.
    self.Window.MessageDialog(messageText = retval);
    self.Window.PurgeEvents();
  end if;
```

Declaring variables for event parameters

To declare the variables to store the event's parameters, you must enter a variable name for each parameter that you wish to receive. You do not need to specify a data type. Forte automatically uses the data type of the corresponding parameter as you declared it in the original event definition. These variables are always local to the individual **when** clause. You cannot use existing variables in this list. If an event parameter has the same name as an existing variable, the event parameter hides the variable.

To indicate how the variables and parameters correspond, you can use the parameter names, the parameter positions, or both.

Parameter names

To use the parameter names, enter each variable name followed by the corresponding parameter name. These can be in any order and you can exclude any parameters you wish.

Example: declaring variables for event parameters

```
-- Using parameter names in an event handler’s when clause
  when <picture_field_for_drop>.ObjectDrop(
      dat = sourceData, dtype = SourceDataType) do
    if dtype = SD_IMAGE then
      -- Must cast because type of ‘dat’ is Object
      picture_field_for_drop = ImageData(dat);
    end if;
  ...
```

Parameter positions

To use the parameter positions, simply enter the variable names in the same order as the parameters in the event definition. This order must be identical to the order in the original definition and you cannot exclude any parameters unless they are at the end of the series.

To use a combination of the two techniques, you must first enter the variable names by position. You can then use parameter names for the remaining variables in any order.

Example: parameters by name and position

```
-- Using parameters in an event handler’s when clause
  when self.Window.Form.AfterMarkLine
      (sx, sy, ey = EndY, ex = EndX) do
    -- ‘sx’ and ‘sy’ are starting coordinates of the line
    -- ‘ex’ and ‘ey’ are the ending.
      ...
```

Class parameters

Remember that for class parameters the variable is a reference to the object not the object itself. Therefore, if you make changes to the object within the event handler, these will be visible to any other data items that reference the object.

Statement block

The statement block provides the code that is executed when the event handler receives the specified event. This can include any TOOL statements except the **return** statement.

exit statement

You can use the **exit** statement within the **when** clause. If there is an enclosing loop statement, the **exit** statement exits from the loop statement. If there is no enclosing loop statement, the **exit** statement exits from the **event** statement that registers the event handler, not just from the **when** clause.

continue statement

The **continue** statement in the **when** clause transfers control to the enclosing loop statement. If there is no enclosing loop statement in the event handler, the **continue** statement transfers control to the **event loop** statement that registers the event handler.

Scoping for variables

The statement block for the **when** clause determines the scope for any variables or constants that are declared within it. Variables and constants that you declare within the statement list are available within the **when** clause only and cannot be accessed by the rest of the event handler.

Exception Handling

The exception handler for the **event handler** statement provides exception handling for the event handler as a whole. See “Exception” on page 113 for details about using exception handlers.

Because the event handler is a separate block of code, any exceptions handled by its **exception** clause affect only the **event handler** statement and do not affect the calling method or **event** statement. If an exception occurs while the event handler is being executed, the **exception** clause of the event handler catches that exception. The calling **event** statement or method will therefore not see that exception and will continue executing.

One thing you need to consider is which part of the event handler is being executed when the event handler catches the exception. This effects whether or not the events in the event handler are registered.

Exceptions in **preregister** clause

If an exception occurs while the **preregister** clause is being executed, the **exception** clause of the event handler catches that exception before the events in the event handler have been registered. Therefore, the events will not be registered in the enclosing event loop. However, the enclosing event loop will still be active and all of its events will still be registered.

Exceptions in **postregister** clause and **when** clause

If an exception occurs while a **when** clause or the **postregister** clause is being executed, the events in the event handler have already been registered, and so will still be registered in the enclosing event loop. The enclosing event loop will still be active and all of its events will also still be registered.

If an exception occurs in any of the event handler clauses, and the **exception** clause of the event handler does *not* handle the exception, the exception is passed to the caller. The caller will then handle the exception or pass it on to its own caller. This is the standard exception handling behavior for TOOL. If the caller is in an **event** statement, the **event** statement will exit and deregister all of its events.

Another issue to consider is whether the caller of the event handler is an **event** statement or a method. This may affect whether or not the unhandled exception causes the enclosing **event** statement to exit. While a **when** clause of the event handler is being executed, the caller of the event handler is always the enclosing **event loop** or **event case** statement. However, the **register** statement can be called from a method that is separate from the loop. Therefore, while the **preregister** or **postregister** clause of an event handler is being executed, the caller of the event handler could be a method. If the caller is a method and the exception is not handled by the event handler, the exception could be handled by the calling method. And, if the exception is handled by the calling method, the exception will not force the enclosing event loop to exit.

When the caller is the **event case** statement, an exception in the **when** clause of an event handler will always cause the **event case** statement to exit. This is true even if the event handler handles the exception, because an **event case** statement always exits after processing one event, even if that processing happens to get an exception.

Interface

The **interface** statement creates an interface.

Syntax

interface *interface_name* [**inherits** [**from**] *super-interface*]
[[**has public**] *component_definitions*]...
[**has property** [**extended** = (*name*=*value*[,*name*=*value*]...)]...
end [**interface**]**;**

Example

```
interface TaxCalculationIFace
has public event TaxCalculated(input theTax: double);
has public method CalculateTax(input theSale: AAInterfaces.Sale)
      : double;
end interface;
```

Description

Typically, you create interfaces using the Interface Workshop. However, the **interface** statement allows you to define interfaces in a file. To import the interface definition from the file into your development repository, you can use the **Compile or Import Class** commands in Fscript or the **Import** command in the Project Workshop.

Interface Name

The interface name provides a name for the new interface. If an interface by that name already exists, your new definition replaces the existing definition.

We recommend that you adopt a naming convention to distinguish your interfaces from the Forte system classes.

Inherits Clause

Specifying the super-interface

The **inherits** clause specifies one interface from which this interface will inherit virtual attributes, methods, events, event handlers, and constants. You can specify any interface, or none; if you specify no interface then this interface inherits no components.

Component Definitions

The components you can define for an interface include:

- virtual attributes
- events
- methods
- event handlers
- constants

Virtual Attributes

The syntax for defining a virtual attribute in an interface is the same as the syntax for defining a virtual attribute in a class.

To define a virtual attribute, you must specify the attribute name, the attribute type, and two expressions that determine the value of the attribute when it accessed or assigned. The syntax is:

virtual [**attribute**] *attribute_name* **:** *type* = (**get** = *expression*,
  [**set** = *expression*]) [**has property** [**extended** = (*name*=*value*[,*name*=*value*]...)]];

See “Virtual Attributes” on page 162 for more information about defining a virtual attribute.

Events

The syntax for defining an event in an interface is the same as the syntax for defining an event in a class.

You define an event by specifying an event name and an optional list of parameters. The syntax is:

event *event_name* [(*name* **:** *type* [= *value*] [, *name* **:** *type* [= *value*]]...)]
  [**has property** [**extended** = (*name*=*value*[,*name*=*value*]...)]];

See “Events” on page 163 for more information about defining an event.

Methods

The syntax for defining a method in an interface is the same as the syntax for defining a method in a class.

To include a method in an interface, you must specify the method name, the method parameters, and the method return type. If you want to create return and exception events to be used with the **start task** statement, you must specify names for these events. The syntax is:

[**method**] *method_ name* [(*parameter_list*)] [**:** [**copy**] *return_type*]
  [**where completion** = (*event_setting* [, *event_setting*])
  [**has property** [**extended** = (*name*=*value*[,*name*=*value*]...)]];

parameter _list is:

([**copy**] [**input** | **output** | **input output**] *name* **:** *type* [= *value*]
  [**,** [**copy**] [**input** | **output** | **input output**] *name* **:** *type* [= *value*]]...)

event_setting is:

return = *event_name* | **exception** = *event_name*

Note that you can overload a method in an interface, but you cannot override a method. Because the interface provides only the method signatures, overriding an inherited method in an interface has no significance.

Event Handlers

The syntax for defining an event handler in an interface is the same as the syntax for defining an event handler in a class.

To include a named event handler in an interface you must specify the event handler name and the event handler parameters. You provide the code for event handler separately, in each class that implements the interface. The syntax is:

event handler *handler_ name* [(*parameter_list*)]
  [**has property** [**extended** = (*name*=*value*[,*name*=*value*]...)]];

parameter _list is:

([**copy**] [**input**] *name* **:** *type* [= *value*]
  [**,** [**copy**] [**input**] *name* **:** *type* [= *value*]]...)

Constants

To define an interface-level constant, use the **constant** statement within the definition list. See "Constant" on page 171 for information.

Has Property Clause

The **has property** clause for an interface allows you to set any number of extended properties for the interface or any of its elements, just as you can for a class (see "Has Property Extended Clause" on page 169). You can assign arbitrary name-value pairs to the interface for whatever purpose you choose. For example, you might wish to use them for comments.

Method

The **method** statement defines a method for a class.

Syntax

method *class*.*method_name* [(*parameter_list*)] [**:** [**copy**] *type*]
begin
  statement_block
  [*exception_handler*]
end [**method**]**;**

Example

```
method Bid.StartBid (
  input bidderName : TextData,
  output paintingForBid : Painting,
  output bidValue : DecimalData,
  output lastBidTime : DateTimeData,
  output bidInProgress : boolean) : TextData
begin
  self.BidInProgress = TRUE;
  self.LastBidder.SetValue(bidderName);
  post self.BidStarted (who = bidderName);
  paintingForBid = self.PaintingForBid;
  bidValue = self.bidValue;
  lastBidTime = self.LastBidTime;
  bidInProgress = self.bidInProgress;
  return bidderName;
end method;
```

Description

The **method** statement provides the body of a method that you create in a **class** statement. This is the code that provides the method’s functionality.

When you use the **method** statement, you must specify the method name, parameters, and return type. These must match the method name, parameters, and return type you specified for the method in the **class** statement.

Class and Method Name

You must specify the class to which the method belongs. Because you can have any number of methods with the same name, Forte uses the class name to assign the method to the class.

The method name is the name you must use to invoke the method. This must be the same as the name you specified for the method in the class definition.

Parameters

The method parameters provide the mechanism for passing values to and from the method. This clause must match exactly with the method's parameter list you specified in the **class** statement. The syntax is:

([**copy**] [**input** | **output** | **input output**] *name* **:** *type* [= *value*]
  [**,** [**copy**] [**input** | **output** | **input output**] *name* **:** *type* [= *value*]]...)

See "Methods" on page 164 for information about the method's parameter list.

Return Type

If a method returns a value, you must specify the type of the returned value. The type for the method specifies the data type of the return value. This clause must match exactly with the return type you specified for the method in the **class** statement. If you specified a return type in the **class** statement, you must also specify it here. See "Class" on page 159 for information about the method's return type.

Statement Block (Method Body)

The statement block is the body of the method. This is where you provide code that operates on the object, invokes methods on other objects, and so on. The statement block can include any TOOL statements.

Note that when the method has a return type, you must use the **return** statement to exit from the method and return a value to the invoking method.

Exception Handler

The optional exception handler provides exception handling for the method as a whole. See "Exception" on page 113 for information about this.

Service

The **service** statement declares a service object.

Syntax

service *name* **:** *class* [= ([*attribute* = *value* [**,** *attribute* = *value*]...])]

Example

```
service MySybase : DBResourceMgr;
service ImageService : ImageMgr =
  (visibility = user);
service AuctionService : AuctionMgr =
  (visibility = environment);
```

Description

Normally you create service objects using the Project Workshop. The **service** statement allows you to define service objects in a file using the Forte Fscript utility.

You can create three types of service objects:

- service objects based on the DBResourceMgr class
- service objects based on the DBSession class
- simple service objects

Each type of service object has its own special set of attributes that you can set when you declare the service object. See below for a description of the attributes that are common to all service objects. This is followed by information on the attributes that are only for specific service objects.

Name

The service object name can be any legal Forte name. It must be unique for the project.

Class

Only certain classes allowed

You can create service objects using custom classes, or some Forte classes. You can use any custom class that has the distributed property set to “allow=on.” However, you can only use certain Forte classes for service objects. Refer to the reference section for any given class, checking the first summary table, to see whether the class can be used for a service object.

To create a DBResourceMgr service object, you use the DBResourceMgr class for the service object’s base class. To define a DBSession service object, you use the DBSession class for the base class. See the manual *Accessing Databases* for information about these classes and creating these types of service objects.

Service Object Attributes

The attributes you can set for any service object are:

Visibility = {**environment** | **user**}
DialogDuration = {**message** | **transaction** | **session**}
Failover = {**TRUE** | **FALSE**}
LoadBalance = {**TRUE** | **FALSE**}
SearchPath = *string*

All of these attributes are optional.

Visibility attribute

The Visibility attribute determines whether the service object is shared between multiple users of the application or whether each user has its own copy of the object. By default, the visibility for a service object is environment. You can use the Visibility attribute to set the visibility to either of the following:

Visibility	Description
environment	The service object is shared between all users in the environment.
user	The service object is not shared. Each user has a separate copy of the object. (User-visible service objects cannot be replicated for load balancing or failover.)

DialogDuration attribute

When you partition your project in the Partition Workshop, you can request replication of your service object to provide fault tolerance or parallel processing (see *Forte 4GL Programming Guide* for information on this). If you plan to do this, you need to specify the interval over which the service object retains state information for its callers. Forte uses this information to determine when to switch to a backup service object (for fault tolerance) or how to route messages (for parallel processing).

The DialogDuration attribute specifies the length of this interval. You can specify the following values:

Value	Definition
message	Does not retain state information.
transaction	Retains state information for the duration of a transaction.
session	Retains state information for the entire session. This is the default value of the DialogDuration attribute.

You must specify the behavior that the creator of the class has implemented for that class. In other words, if the service object will retain state information on behalf of the user for longer than a message, you cannot specify message duration. If the service object will retain state information for longer than a transaction, you must specify session duration.

DialogDuration attribute for Session service objects

For session service objects, the dialog duration must be greater than or equal to the dialog duration you specified for the resource manager that the session is associated with.

Failover attribute

The Failover attribute specifies whether or not the service object will be replicated to provide fault tolerance. When you partition the project, you specify the actual number of startup replicates to be provided for failover.

LoadBalance attribute

The LoadBalance attribute specifies whether or not the service object will be replicated for parallel processing. When you partition the project, you specify the actual number of startup replicates to be provided for load balancing. The LoadBalance attribute can be set to TRUE only if the DialogDuration attribute is set to message or transaction.

SearchPath attribute

The environment search path for the service object specifies the connected environments in which Forte searches for failover service objects and for service objects in reference partitions. This is the default search path for the service object.You can override the default search path for a service object in the Partition Workshop by specifying a search list for the service object within a particular configuration (see *A Guide to the Forte 4GL Workshops* for information).

To specify the environment search path, enter a string that includes one or more environment paths. Forte searches for the service object in the same order as the paths in the string.

The syntax of the search list string is:

path [(**a**)] [**:** *path* [(**a**)]...

path is:

(**%** | **%***environment_name*)

A special (a) option allows you to specify that the service object identified by a specific path should automatically be started if necessary.

You can use an environment variable to specify an environment name. The value for the environment variable is set on first access to the service object, using the value of the environment variable as set on the service object's partition. The syntax is:

${*environment_variable_name*}

Be sure to include the braces!

The following example illustrates a search list that looks first in the current environment, second in the "la" environment, and last in the "sf" environment:

```
%:%la:%sf
```

DBResourceMgr Service Object Attributes

The attributes you can set for a DBResourceMgr service object are:

Visibility = {**environment** | **user**}
DialogDuration = **session**
Failover = {**TRUE** | **FALSE**}
LoadBalance = {**TRUE** | **FALSE**}
SearchPath = *string*
ExternalManager = *string*

All of these attributes are optional, except the ExternalManager attribute, which is required.

ExternalManager attribute

For a DBResourceMgr service object, you must specify the external resource manager associated with the service object. Forte uses this resource manager for the service object unless you override this setting for a particular configuration.You can specify the name of any database resource manager that is defined in your environment. For more information on resource managers, refer to the manual *Accessing Databases*.

DBSession Service Object Attributes

The attributes you can set for a DBSession service object are:

Visibility = {<u>**environment**</u> **| user**}
DialogDuration = {**transaction |** <u>**session**</u>}
Failover = {**TRUE | FALSE**}
LoadBalance = {**TRUE | FALSE**}
SearchPath = *string*
ExternalManager= *string*
ResourceName= *string*
UserName = *string*
UserPassword = *string*

The ExternalManager, ResourceName, UserName, and UserPassword attributes are required. All other attributes are optional.

ExternalManager attribute

For a DBSession service object, you must specify the external resource manager associated with the service object. Forte uses this resource manager for the service object unless you override this setting for a particular configuration. You can specify the name of any database resource manager that is defined in your environment. For more information on resource managers, refer to the manual *Accessing Databases*.

ResourceName attribute

The ResourceName attribute specifies the name of the specific database that you wish to access.

UserName and UserPassword attributes

The UserName and UserPassword attributes specify the user name and password for the particular database session. These are the same as the corresponding parameters on the Connect method of the DBSession class. See *Accessing Databases* for information.

Simple Service Object Attributes

The attributes you can set for a simple service object are:

Visibility = {**environment | user**}
DialogDuration = {**message | transaction | session**}
Failover = {**TRUE | FALSE**}
LoadBalance = {**TRUE | FALSE**}
SearchPath = *string*
public_attribute = *value* [, *public_attribute* = *value*]...

All of these attributes are optional.

Setting initial values

For simple service objects, you can specify initial values for any of the public attributes in the object. To specify the values, enter a list of attribute/value assignments. The value for an attribute can be any constant that is compatible with the data type of the attribute. If the attribute has a class as its type, use an object constructor to specify values for one or more of its attributes. You can use named constants to specify the attribute values (if they are defined as part of the project), but you cannot use variables or attributes.

Any public attributes for which you do not specify a value will be set to the default value for their datatype.

Appendix A

Reserved Words

This appendix contains two lists:

- TOOL reserved words
- SQL reserved words

You should avoid using SQL reserved words for attribute and variable names.

TOOL Reserved Words

and	method
attribute	new
begin	nil
case	not
changed	of
class	of
constant	output
continue	post
copy	postregister
cursor	preregister
do	private
else	property
elseif	public
end	raise
enum	register
event	return
exception	service
exit	sl
false	start
for	struct
forward	super
from	task
handler	then
has	to
if	transaction
implements	true
in	typedef
includes	union
inherits	virtual
input	when
interface	‘where
is	while
loop	

The TOOL key word “application” is not reserved. You can use it for projects, classes, attributes, methods, and so on; however, if you do, you will also need to use dot notation to access the object normally accessed by the key word application.

For example, you would have to type Framework.application instead of just “application” if the current class had an attribute or method with the name application.

SQL Reserved Words

all	group
any	having
as	immediate
asc	insert
between	into
by	like
close	minus
connect	null
current	on
default	open
delete	order
desc	procedure
distinct	raise
escape	revoke
execute	select
exists	session
extend	set
extent	some
fetch	unique
fragment	update
from	values
grant	where

Appendix B

Forte TOOL Example Applications

Forte provides a number of example applications that illustrate how to use the features described in this manual. 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 Forte Workshops to see how it is implemented. You can modify the examples if you wish.

How to Install Forte Example Applications

You can access the Forte example applications only if they have been installed into your central repository or into a private local repository during installation of Forte, or if you have imported them into your repository.

The examples are located in subdirectories under the FORTE_ROOT/install/examples directory. The example applications are stored as .pex files. If they are not already installed in your repository, import them by including the tstapps.fsc script in Fscript. The tstapps.fsc script is located in the FORTE_ROOT/install/examples/install directory. Bring up Fscript in standalone mode and issue the following command:

```
fscript> UsePortable
fscript> SetPath %{FORTE_ROOT}/install/examples/install
fscript> Include tstapps.fsc
```

This will import most of the example applications and quit Fscript. Note that certain highly specialized examples are not automatically imported by tstapps.fsc.

To run an application, select it in the Repository Workshop's plan browser and then click on the Run button.

If you want to remove all the examples from your workspace, you can do so by including the remprj.fsc script in Fscript. Bring up Fscript in standalone mode and issue the following commands:

```
fscript> UsePortable
fscript> SetPath %{FORTE_ROOT}/install/examples/install
fscript> Include remprj.fsc
```

This will exclude all the example applications and quit Fscript.

Overview of Forte TOOL Example Applications

This section provides an overview of the Forte TOOL example applications. For a complete list of the Forte example applications, see *A Guide to the Forte 4GL Workshops*.

The margin note for each of the following tables shows the name of the subdirectory under FORTE_ROOT/install/examples where you can find the .pex files for the examples. For the complete description of an individual application, see "Application Descriptions" on page 196, which lists the applications in alphabetical order.

TOOL Examples

tool/

Example	Description
Auction	Illustrates prominent features of a Forte distributed application.
AuctionServerProject	Acts as server to the Auction project.
ImageProject	Provides images for the Auction project.
ImageTester	Retrieves an image using the ImageProject service.

Application Descriptions

This section lists the example applications in alphabetical order. Each example has five sections describing it.

The **Description** section defines the purpose of the example, what problem it solves, and what TOOL features and Forte classes it illustrates.

The **Pex Files** section gives you the subdirectory and file names of the exported projects. The examples are in subdirectories under the FORTE_ROOT/install/examples directory. You can 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. Import them in the order given so that dependencies will be satisfied.

The **Mode** section indicates whether the application can be run in either standalone or distributed mode, or whether it must be run in distributed mode.

The **Special Requirements** section identifies whether you need a database connection, an external file, or any other special setup.

Finally, the **To Use** section tells you how to step through the application's functions.

See the *Forte 4GL System Management Guide* if you need directions to set up a Forte server. See *Accessing Databases* if you need information on how to make a connection to a database. The database examples run against either Sybase or Oracle.

Auction

Description Auction illustrates prominent features and capabilities of a Forte distributed application: GUI independence, distributed processing, event handling, multitasking, and image handling. The application allows a number of bidders located at their respective computers to bid on a set of paintings being offered by an auctioneer located at some other computer. The Art Auction application provides a list of paintings available for bidding and notifies interested bidders when a price changes.

Pex Files frame/utility.pex, tool/imageprj.pex, tool/aucserv.pex, tool/auction.pex.

Mode Standalone or Distributed.

Special Requirements The image files used by this application must be located in $FORTE_ROOT/install/examples/images.

▶ **To use Auction:**

1. Start up the auction by clicking the Be Auctioneer option in the radio list, then clicking the Start Auction button.
2. Assume the role of a bidder by clicking the Be Bidder option in the radio list. You should click on a painting in the array, then click the View Painting button.

   From the painting window, you can double-click on the image to see it enlarged. You can also click the Bid button to set a bid.
3. Another bidder can view available paintings being offered and then join the bidding process.

   Both bidders become involved in bidding on the same painting. In the standalone use of this application, you can simulate a second bidder on the same screen by opening a second bidding window.

ImageTester

Description ImageTester retrieves an image using the ImageProject service. It is normally used to start up the ImageProject service in conjunction with a demonstration of the Auction application. It can also be used to set up a reference partition in conjunction with Auction.

Pex Files frame/utility.pex, tool/imageprj.pex, tool/imagetst.pex.

Mode Standalone or Distributed.

Special Requirements The image files used by this application must be located in $FORTE_ROOT/install/examples/images.

▶ **To use ImageTester:**

1 Enter a bitmap graphic file name in the Name field and click on the GetImage button.

Graphic files, for example mona.fso, can be found in FORTE_ROOT/install/examples/images. The .fso suffix is automatically appended. Enter the filename without the suffix. For example, enter 'mona'.

Index