CHAPTER 5

EGL System Resources

This chapter reviews three EGL resources that are available when you write functions: statements, system libraries, and a mechanism for exception handling.

Our focus remains on core EGL functionality. The help system for Rational Business Developer provides details on other areas, including the extensive EGL support for the data-access technologies DL/I and WebSphere MQ; for the transaction managers CICS and IMS; and for the user-interface technologies described as Text UI, Console UI, and Web transactions.

EGL Statements

In this section, we describe EGL statements for general use. In the next chapter, we lists statements that access files and relational databases.

Table 5-1 list the statements covered here.

Table 5-1: Selected EGL Statements for General Use

Statement Type	Purpose
assignment	To assign the value of an expression to a data area
call	To transfer control to another program
case	To run one of several alternative sets of statements

Table 5-1: Selected EGL Statements for General Use (Continued)

Statement Type	Purpose
comment	To document code
function invocation	To run a function
continue	To return control to the start of an enclosing loop in the same function
exit	To end the processing of the nearest enclosing statement of a stated kind
for	To define a block of statements that run in a loop until a specified value is reached
forward	To transfer control from a JSF handler to (in most cases) a Web page
if, else	To define a block of statements that run if and only if a specified condition applies; also, to define an alternative block
move	To copy data from a source to a target, with processing not available in assignment statements
return	To return control from a function and optionally to return a value to the invoker
set	To establish characteristics of a record or record field.
transfer	To transfer control from one main program to another
while	To define a block of statements that run in a loop until a test fails

Code Documentation

A comment lets you add documentation to your code and is useful almost anywhere in an EGL source file, not only in functions.

A single-line comment begins with double slashes (//), as shown next, in boldface.

```
i INT = 2;

// a while loop follows
while(i > 0)
   sysLib.writeStdOut(i);
   i = i-1
end
```

A comment that can span multiple lines begins with a slash and asterisk (/*) and ends with an asterisk and slash (*/), as shown next.

```
i INT = 2;

/* a while loop
   does not follow */
sysLib.writeStdOut(i);
```

Data Assignment

You assign data by coding the assignment, **move**, and **set** statements.

Assignment

The primary data-assignment statement copies an expression into a data area. In the following example, **fullString** receives the value *Welcome to EGL*.

```
oneString, fullString STRING;
oneString = "Welcome ";
fullString = oneString + "to EGL";
```

Move

The **move** statement copies data from a source area to a target area, in any of three ways. First, byte by byte; second, by matching the field names in the target and source; and third, by matching the field positions in the target and source. The **move** statement provides additional options for copying values from one array to another.

Here are two Record parts.

```
Record MyCustomerOneType type BasicRecord
   ID INT;
   title STRING;
end

Record MyCustomerTwoType type BasicRecord
   age INT;
   title STRING;
end
```

Here are two declarations.

```
myCustomer01 MyCustomerOneType;
myCustomer02 MyCustomerTwoType;
```

The effect of **move** by name is to copy the values only for fields whose names match. The result of the next example is to copy a value from one field named **title** to another.

```
myCustomer01.ID = 25;
myCustomer01.title = "Doctor";
myCustomer02.age = 40;
myCustomer02.title = "Officer";
move myCustomer01 to myCustomer02 byname;

sysLib.writeStdOut(myCustomer02.age);  // 40
sysLib.writeStdOut(myCustomer02.name); // Doctor
```

In contrast, the effect of **move** by position is to copy the values for each field in a record, without regard to the field names. .

```
myCustomer01.ID = 25;
myCustomer01.title = "Doctor";
myCustomer02.age = 40;
myCustomer02.title = "Officer";
move myCustomer01 to myCustomer02 byposition;

sysLib.writeStdOut(myCustomer02.age);  // 25
sysLib.writeStdOut(myCustomer02.name); // Doctor
```

The last example is with arrays. The assumption in each of the following cases is that the target array has the number of elements needed to accept the content being copied.

```
// move "Buy" to elements 2, 3, and 4 in temp
move "Buy" to temp[2] for 3;

// move elements 2, 4, and 4 from temp
//      into elements 5, 6, and 7 in final
move temp[2] to final[5] for 3;
```

Set

The **set** statement establishes characteristics of a record, form, or field. The statement has many variations. For example, in relation to a record, you can reset the field values to those initially specified in the Record part definition. The boldface statement in the following code has that effect.

```
Record DepartmentPart type BasicRecord
   Department STRING = "Sales";
   BudgetCode INT;
end
Function MyFunction()
   MyDept DepartmentPart;
   SysLib.writeStdOut(MyDept.Department);
   MyDept.Department = "Marketing";
   SysLib.writeStdOut(MyDept.Department);
   set MyDept.Department initial;
   SysLib.writeStdOut(MyDept.Department);
end
```

The code writes **Sales** and then **Marketing** and then **Sales**.

Conditional Processing

The **if** and embedded **else** statements provide conditional processing, as does the **case** statement.

If, Else

The **if** statement defines a block of statements that run if and only if a specified condition applies. The **else** statement defines an alternative block of statements.

The following code sets msgText in various cases. If msgStatus equals 1, msgText is set to *Yes!*; if msgStatus equals 0, msgText is set to *No!*; and otherwise, msgText is set to *Service invocation failed*.

```
if (msgStatus == 1)
   msgText = "Yes!";
else
   if (msgStatus == 0)
      msgText = "No!";
   else
      msgText = "Service invocation failed!";
   end
end
```

Case

The **case** statement runs one of several alternative sets of statements.

You can specify a value for comparison, as shown next.

```
case (msgStatus)
   when(1)
      msgText = "Yes!";
   when(0)
      mgText = "No!";
   otherwise
      msgText = "Service invocation failed!";
end
```

In this example, if **msgStatus** evaluates to 1, **msgText** receives the value *Yes!*; if **msgStatus** evaluates to 0, **msgText** receives *No!*; and if **msgStatus** evaluates to any other value, **msgText** receives *Service invocation failed!*

The **case** statement runs all statements in a given **when** or **otherwise** clause, and control never passes to more than one clause. In the next example, if

myCode evaluates to 1, the functions **myFunction01** and **myFunction02** run and the others do not.

```
case (myCode)
  when (1)
     myFunction01();
     myFunction02();
  when (2, 3, 4)
     myFunction03();
  otherwise
     myDefaultFunction();
end
```

If you don't specify a value for comparison, the **case** statement runs the first clause for which a condition resolves to *true*. In the following example, if **myCode** evaluates to 4, the function **myFunction03** runs.

```
case
  when (myCode == 3)
     myFunction01();
  when (myCode > 3)
     myFunction03();
  otherwise
     myDefaultFunction();
end
```

Loop Control

The **for**, **while**, and **continue** statements provide loop control.

For

The **for** statement defines a block of statements that run in a loop until a specified value is exceeded. For example, the following code writes the numbers 10, 20, 30, and 40 to the standard output.

```
for (i int from 10 to 40 by 10)
   sysLib.writeStdOut(i);
end
```

The following code writes the numbers 40, 30, 20, and 10.

```
for (i int from 40 to 10 decrement by 10)
   sysLib.writeStdOut(i);
end
```

While

The **while** statement defines a block of statements that run in a loop until a test fails. The following code writes the numbers 10, 20, 30, and 40.

```
i INT = 10;
while (i <= 40)
  sysLib.writeStdOut(i);
  i = i + 10;
end
```

Continue

The **continue** statement returns control to the start of an enclosing loop in the same function. For example, the following code writes the numbers 1, 2, and 4 to the standard output.

```
for (i int from 1 to 4 by 1)
  if (i == 3)
    continue;
  end
  sysLib.writeStdOut(i);
end
```

Transfer of Control Within a Program

Function invocations and the **return** and **exit** statements transfer control within a program or handler.

Function Invocation

A function invocation runs a function and includes arguments to match the function parameters. The arguments must match the parameters in number, with some variation allowed in the data type, as specified in compatibility rules.

The next statement passes a string to a function.

```
myFunction("test this string");
```

If a function returns a value, two rules apply. First, you can code a variable to receive that value from the function, but the variable is not required. Second, you can code the function invocation inside a larger expression. For example, if the function in the next expression returns the name *Smith*, the string sent to the standard output is *Customer 23 is Smith*.

```
sysLib.writeStdOut("Customer 23 is "
                  + getCustomerName(23));
```

Return

The **return** statement returns control from a function and optionally returns a value to the invoker. In the next example, the statement returns 0.

```
for (i int from 10 to 40 by 10)
   sysLib.writeStdOut(i);
end
return(0);
```

Exit

The **exit** statement ends the processing of the nearest enclosing statement of a stated kind. For example, the following code stops processing the **for** statement when the value if i is 3.

```
for (i int from 1 to 4 by 1)
  if (i == 3)
    exit for;
  end
  sysLib.writeStdOut(i);
end
```

The code writes the numbers 1 and 2 to the standard output.

Transfer of Control Out of a Program

The **call**, **forward**, and **transfer** statements transfer control to logic that's outside the program or handler.

Call

The **call** statement transfers control to another program. The arguments must match the program parameters in number and type.

```
myCustomerNumber = 23;
myCustomerName = "Smith";
call myProgram(myCustomerNumber, myCustomerName);
```

Forward

The **forward** statement in a JSF handler transfers control; in most cases, to another Web page.

```
forward to "myWebPage";
```

The current example forwards control to the logic that the JavaServer Faces runtime identifies as *myWebPage*. In most cases, that logic is (essentially) a Web page. We cover JavaServer Faces in later chapters.

Transfer

The **transfer** statement transfers control from one main program to another, ending the first program and optionally passing a record. The following code transfers control to **Program02** and passes a record named **myRecord**.

```
transfer to Program Program02 passing myRecord;
```

System Libraries

In addition to developing libraries that include variables, constants, and functions, you can access EGL system libraries such as the following ones.

- The **sysLib** library lets you write to an error log or a standard location, commit or roll back database changes, retrieve properties and messages from text files, wait for time to elapse, or run an operating-system command.
- The **strLib** library lets you format date and time variables and manipulate strings.
- The **mathLib** library lets you perform common mathematical and trigonometric operations. You can round a number in various ways and determine the maximum or minimum of two numbers.
- The **datetimeLib** library lets you retrieve the current date and time and lets you process dates, times, and intervals in various ways.
- The **serviceLib** library lets you specify a service location to be accessed at run time.
- The **lobLib** library lets EGL-generated Java code work with variables of type BLOB (binary large object) or CLOB (character large object). You can associate a file with a variable of one of those types, transfer data to and from the file, and gain access to a string that represents the data.

Some libraries are specific to a runtime technology; for example:

- The **sqlLib** library lets you interact with relational database management systems; for example, to connect to a database at run time.
- The **j2eeLib** library lets you interact with a Web application server from an EGL JSF handler.

Exception Handling

EGL-generated code can encounter an exception during the following runtime operations: a program call or transfer; access of a service or library; a function invocation; access of persistent storage; a data comparison; or a data assignment. You also can *throw*—register—an exception in response to some runtime event; for example, a user's entering an invalid customer ID at a Web browser.

To make an exception available to be *caught*—that is, to be handled—you embed business logic inside a **try** block. Here's an outline of a **try** block.

```
try
   // place your business logic here

   onException
      (exceptionRecord ExceptionType01)
      // handle the exception here

   onException
      (exceptionRecord ExceptionType02)
      // handle the exception here
end
```

The **try** block includes zero to many **onException** blocks. Each **onException** block is essentially an error handler and is similar to a function that accepts a single parameter—an exception record—and returns no value. Unlike a function, the **onException** block has no **end** statement; instead, the block ends at the start of the next **onException** block, if any, or at the bottom of the **try** block.

The use of a **try** block has a performance cost, so you may decide to embed only the most exception-prone code in such a block.

In the following example, the attempt to add content to an uninitialized array causes an exception that we describe with the phrase "**NullValueException**."

```
Program myProgram type BasicProgram
   myStringArray STRING[];
   Function main()
      try
         myStringArray.appendElement("One"); // error
         onException (exception NullValueException)
            myStringArray = new STRING[];
            sysLib.writeStdErr ("NullValueException");
         onException (exception AnyException)
            sysLib.writeStdErr ("AnyException");
      end

      sysLib.writeStdErr
         ("Size of array is " +
            myStringArray.getSize());
   end
end
```

In this example, an **onException** block catches the exception, initializes the array, and writes the name of the exception type to the standard error output. A statement outside the **try** block then writes the following string to that output: *Size of array is 0*.

In general, the exception is caught by the first **onException** block that is specific to the exception type; here, the type is NullValueException. However, if no onException block is specific to the exception type, the EGL runtime invokes the **AnyException** block, if any. The placement of the **AnyException** block in the list of **OnException** blocks is not meaningful; the **AnyException** block is invoked only as a last resort.

An **OnException** block can itself include try blocks, to any level of nesting. In the following example, a statement in a **try** block also throws a **NullValueException**.

```
Program myProgram type BasicProgram
   myStringArray STRING[];
   myStringArray02 STRING[];
   Function main()
      try
         myStringArray.appendElement("One");
         onException (e01 NullValueException)
            try
               myStringArray.appendElement("One");
               onException
                  (e02 NullValueException)
                  myStringArray = new STRING[];
                  myStringArray.appendElement("One");
            end
      end
      sysLib.writeStdErr
         ("Size of array is " +
         myStringArray.getSize());
   end
end
```

The last statement in the example writes the string *Size of array is 1*.

We say that an exception is *cleared* if the code continues running without being interrupted again by that exception. The exception is cleared in the following two cases: the exception causes invocation of an **OnException** block at the current nesting level; or the exception occurs in a **try** block that has no **OnException** handlers at all. That second case has little practical value because you probably don’t want your code to continue running unless you first correct the error or at least log the details.

The function ends immediately in the following case: an exception occurs and the **try** block at the current nesting level includes **onException** blocks, but none of the blocks catches the exception.

Propagation

The next example of error handling is similar to an earlier one, but the business logic in this case invokes the function **appendToArray**, which in turn invokes the function **doAppend**.

```
Program myProgram type BasicProgram
   myStringArray STRING[];
   Function main()
      try
         appendToArray("One");
         onException (exception NullValueException)
            myStringArray = new STRING[];
            sysLib.writeStdErr ("NullValueException");
      end

      sysLib.writeStdErr
         ("Size of array is " +
            myStringArray.getSize());
   end
   Function appendToArray (theInput STRING IN)
      doAppend(theInput);
      sysLib.writeStdErr ("You won't see the message." );
   end
   Function doAppend (theString STRING IN)
      myStringArray.appendElement(theString);
      sysLib.writeStdErr ("You won't see the message." );
   end
end
```

Listing 5.1: Propagation of exception

The **NullValueException** exception now occurs in **doAppend**, but the exception is handled as before, in the **main** function.

Figure 5.1 illustrates the general rule.

If a function throws an error that is not cleared, the function immediately ends, and the exception *propagates*—moves its influence—to the function's invoker, which clears the error or immediately ends. An uncleared exception propagates to the immediate invoker and then to progressively higher-level invokers until the exception is cleared or the program ends.

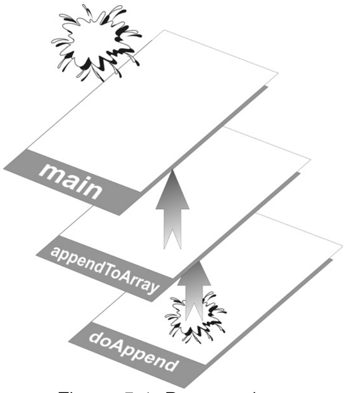

Figure 5.1: Propagation

An exception also propagates from a called program to progressively higher-level callers. However, a failed service returns ServiceInvocationException, regardless of the exception that caused the service failure.

Exception Fields

Two fields—**message** and **messageID**—are available in every exception record. In relation to an exception that is defined by EGL, the **message** field contains a series of messages, each with an error number, and the **messageID** field contains the error numbers alone. For example, we might have coded one of our earlier **OnException** blocks as follows.

```
onException (exception NullValueException)
   myStringArray = new STRING[];
   sysLib.writeStdErr
      ("Handled this issue: \n" + exception.message);
```

In EGL-generated Java code, the new-line character (**\n**) makes the first error message appear on a new line, as shown next.

```
Handled this issue:
EGL0098E The reference variable named myStringArray is null.
EGL0002I The error occurred in the myProgram program.
```

Our next example shows a way to throw an exception of your own. We begin with an SQL Record part and an Exception part.

```
Record MyCustomerRecord type SQLRecord
   { keyItems = [customerNumber],
     tableNames = [["Customer"]] }
   customerNumber STRING;
   creditScore INT;
end

Record CustomerException type Exception
   customerID STRING;
end
```

After we declare a record that is based on the SQL Record part, we assign a customer number and code an EGL **get** statement to retrieve details about the customer identified by that number. The logic is similar to what is shown here.

```
myCustomer MyCustomerRecord;
myCustomer.customerNumber = "A1234";
get myCustomer;
```

We ignore the need to handle an SQL exception but show how to throw an exception of your own. In the following program, the CustomerException block uses the following fields: **customerID**, which was defined explicitly in the CustomerException part, and **message**, which is of type STRING and is present in any Exception record.

```
Program myProgram type BasicProgram

   Function main()
      myCustomer MyCustomerRecord;
      myCustomer.customerNumber = "A1234";
      try
         retrieveOne(myCustomer);
         onException (exception CustomerException)
            sysLib.writeStdErr
               ("Customer: " + exception.customerID
                + "\nIssue: " + exception.message);
     end
   end
```

Listing 5.2: Throwing an Exception (part 1)

```
   Function retrieveOne(theCustomer MyCustomerRecord)
      get theCustomer;
      if (theCustomer.creditScore < 310)
         throw new CustomerException
            { customerID = theCustomer.customerNumber,
              message = "No Credit" };
      end
  end
end
```

Listing 5.2: Throwing an Exception (part 2)

After the get statement runs, the record **theCustomer** includes the credit score for the customer whose number is A1234. If the score is less than 310, the **throw** statement creates a new record, initializing the record fields for use in the **CustomerException** block. The output is as follows.

```
Customer: A1234
Issue: No Credit
```