A Programmer's Guide to the New Exception-Handling Concept in ABAP

Gerd Kluger and Christoph Wedler

Gerd Kluger, Business Programming Languages Group, SAP AG

Christoph Wedler, Business Programming Languages Group, SAP AG

(complete bios appear on page 50)

Errors happen. There are few things more certain than that in software development. We, as software developers, must account for this inevitability in our programs and determine what should happen once an error has occurred. Must the program terminate or is there a way to recover? And what is the best way to specify handler coding for specific errors?

In modern programming languages there is a concept that addresses these issues. This concept is called *exception handling*. An exception is an event during the execution of a program that interrupts the normal flow of control. An exception-handling mechanism makes it possible to react to these events.

In this article, we present a new exception concept in ABAP that has been introduced with Release 6.10. We will provide detailed explanations on how to work with these new exceptions and compare them to existing error-handling capabilities in ABAP.[1]

Since the new exception concept is based on object-oriented (OO) features, you should have some basic knowledge of OO principles. You can find an especially good overview in the book *ABAP Objects: An Introduction to Programming SAP Applications* by Horst Keller and Sascha Krüger (Addison-Wesley, 2002).

[1] Note that some of these existing error-handling capabilities are, unfortunately, also called "exceptions." To distinguish between the two throughout this article, we use the term *class-based exceptions* for the new ones and *classical exceptions* for the existing ones. If we simply talk about "exceptions," unless noted otherwise, we mean "class-based exceptions."

What's Wrong with the Classical Exceptions?

If there are already exceptions existing in ABAP, then why is there a need for a new exception concept? In the existing concept, exceptions can be raised via the `RAISE` statement, and function modules or methods then declare them via the `EXCEPTIONS` clause. To handle them, when calling the function module or method you can define a mapping from each exception's name to a number, and then check the `sy-subrc` for one of these numeric values once the function module or method returns.

As an ABAP programmer, you probably know how cumbersome it is to work with classical exceptions, since you must handle *all* of them *immediately* after a function or method call. It is often the case, however, that instead of handling all exceptions, you will want to select only certain exceptions for handling and pass on the rest to the caller. It is possible to do this with classical exceptions, but it's a rather tedious process since you have to do the mapping manually, exception by exception. Besides being laborious, these mappings clutter your code and make it difficult to read — if you look at the coding there is no clear separation between the regular coding that adds to the functionality of your application and coding that is there for error-handling purposes.

Another issue is the grouping of exceptions. When you define the mapping between exceptions and numbers you can group all exceptions, that are not explicitly mapped, to a single group called `others`. But beyond that, there is no way to deal with similar exceptions, like there is with runtime errors, which can be handled by `CATCH SYSTEM-EXCEPTIONS`. There you have error groups like `arithmetic_errors` and `conversion_errors`, but you cannot define your own error groups.

Then there is the issue of control flow, or lack thereof. If you look closely at what happens when a classical ABAP exception occurs, you will notice that an actual change in control flow, one of the key features of exceptions, does not take place. The values returned are merely return codes. It is desirable to have a real change in control flow, even across multiple levels of the call hierarchy. Programmers should be able to easily handle exceptions they are capable of dealing with, and the rest should be propagated automatically to the next level in the call hierarchy.

Finally, a classical exception is just a signal that a problem has occurred. There is no additional information accompanying the exception, such as information about the context in which the exception occurred (e.g., its location). This is probably not crucial for old exceptions since handlers tend to be close to the location of the exception (due to the need to handle exceptions immediately, as stated earlier). In this case, the handler will most likely know about the context, e.g., the parameter values. But if there were exceptions that could change the flow of control, thereby making it possible for handlers to be far away from the place where a problem occurred, the ability to transport additional information to the handler would become very important.

With Release 6.10, SAP introduced a completely new exception concept in ABAP to overcome these shortcomings. While influenced by languages like C++ and Java, the ABAP exception concept has some outstanding features of its own that make it unique amongst all other languages.

Main Features of the New Exception Concept

Exceptions in 6.10 are represented by objects that are instances of classes. Each such class — called an *exception class* — describes one typical exceptional situation, e.g., a division by zero, an overdrawn account, etc. The exception object is the occurrence of such a situation at some specific point during runtime.

This new approach solves many of the shortcomings programmers now wrestle with:

- ☑ Attributes can store additional information about the cause of the exception situation. Since the object can store any data, exception handlers need no longer know the exception context; they can simply get it from the exception object.

- ☑ Grouping is facilitated via inheritance. Every class is a natural candidate to represent all of its subclasses. There is one common ancestor to all exception classes; it is called `cx_root`. To carry it to the extreme, you could use this class to define handlers for all types of exceptions.

- ☑ Inheritance enables refinement of exceptions — e.g., by adding new attributes — so you can reuse existing exceptions by making them more specific.

The key principle of exception handling is the change in control flow. The raising of an exception thus comprises:

1. The creation of an exception object.

2. The propagation of this object along the call chain until a suitable handler is found.

If no handler is found, a runtime error will occur. Because you can store context information in the exception object, a detailed description of the error can be given, as you will see later in the article.

Raising and Handling Exceptions

Exceptions are raised to indicate that some exceptional situation has occurred. Generally, an exception handler tries to repair the error, find an alternative solution, or, where this is impossible, bring the system to a consistent state and then pass on the error. If the system cannot find a handler for an exception throughout the entire call hierarchy, the program terminates with a short dump.

Raising Exceptions

Exceptions can be *raised* at any point — in a method, a function module, a subroutine, and so on. There are two ways a class-based exception can be raised:

1. As a result of an error detected by the ABAP runtime system — that is, the system raises an exception while it is processing ABAP statements.

2. When an ABAP programmer explicitly raises an exception to signal an error situation in his or her coding.

The simple arithmetic statement below offers an example of an exception raised by the ABAP runtime system:

```
x = 1 / 0.
```

For this statement, where we're trying to divide by zero, the runtime system will raise an exception of type `cx_sy_zerodivide`.

The syntax for raising an exception from within your ABAP code is `RAISE EXCEPTION`.[2] There are two different syntactical variants that you can use to invoke this statement. You can:

a. Raise and create an exception object simultaneously.

b. Raise an exception with an exception object that already exists.

In the first scenario, use the syntax:

```
RAISE EXCEPTION TYPE exc_class
[ EXPORTING a1 = ... b1 = ... ].
```

[2] Raising class-based exceptions differs from raising classical exceptions in that the former uses the `EXCEPTION` addition with the `RAISE` keyword.

After RAISE EXCEPTION, you use the TYPE option with the name of the exception class from which the exception object is to be created. Note that this syntax is close to the CREATE OBJECT statement. You can pass values to the exception class constructor using exporting parameters. The constructor then fills the object attributes with values, which the handler can access at a later time.

The second, less common scenario occurs when an exception object already exists. For example, it may have been created explicitly using a CREATE OBJECT statement. Alternatively, an exception may already have been caught, but after checking some of the attributes, the handler may have realized that it could not deal with this exception properly, and thus decided to raise it again. The syntax in this case is simply:

```
RAISE EXCEPTION ex.
```

Note that ex must be an object variable bound to an exception object instance.

Catching and Handling Exceptions

Handlers are used to "catch" class-based exceptions. Handlers are defined for statements in a TRY block. Each exception that is raised between the entry and exit points of this block can be caught by an appropriate handler.

The TRY block, together with the handlers, is defined by a new syntactical construct — the TRY ... ENDTRY construct — which starts with the keyword TRY and ends with ENDTRY. Within the TRY ... ENDTRY construct, you add handlers by specifying CATCH clauses, which are initiated by the keyword CATCH. The TRY block comprises all statements between TRY and the first CATCH.

A handler consists of all statements between its CATCH clause and the CATCH clause of the next handler (or ENDTRY, if there are no more handlers). The CATCH clause starts with the keyword CATCH, followed by the names of the exception classes of the exceptions to be caught. These may be followed in turn by the optional INTO addition containing the name of a variable for the exception object. This addition is optional since you might be interested only in the *indication* that a certain type of exception has occurred, not in the exception object itself. The statements directly following the CATCH clause are executed only if the handler catches an exception. Each exception is caught only once, by the first suitable handler. A handler is "suitable" if its CATCH clause lists either the class of the exception or a superclass of it.

In the example shown in **Listing 1**, if an exception of the class cx_my1 is raised in line 1, neither of the two handlers will catch it since it occurs before the flow of control has entered the TRY block. If an exception of the class cx_my2 is raised in line 3, the handler in lines 8-10 will catch it. This happens because cx_my2, like all other exception classes, is a subclass of cx_root, and because the handler in lines 5-7 does not catch it. If an exception of the class cx_my3 is raised in the method m3, but is not caught there, the handler in lines 5-7 will catch it. Once the control flow reaches the end of the handler,[3] the runtime system continues executing the program after the keyword ENDTRY in line 11. The same applies if no exception is raised within the TRY block. Thus, after executing the statement in line 4, the system continues after line 11.

This example shows that:

- ☑ Each handler can catch exceptions from more than one class. It does this either explicitly, in that the developer adds more than one exception class (as for the first handler in lines 5-7), or implicitly, in that when the developer adds an exception class, all the exceptions of its subclasses are caught as well. In the extreme case

[3] The end of the handler might not be reached if there is a statement in the handler that changes the control flow by itself — e.g., RETURN or the occurrence of another exception.

Listing 1: Catching an Exception for Handling

```
 1 CALL METHOD o1->m1.      "Can raise an exception of the class cx_my1
 2 TRY.
 3     IF ... . RAISE EXCEPTION TYPE cx_my2. ENDIF.
 4     CALL METHOD o1->m3.  "Can raise an exception of the class cx_my3
 5   CATCH cx_my1 cx_my3.
 6     "Handler of exceptions of the class cx_my1 (and subclasses)
 7     "--- any number of statements ---
 8   CATCH cx_root.
 9     "Handler of all the other exceptions
10     "--- any number of statements ---
11 ENDTRY.
```

mentioned earlier, all exceptions are caught by providing only the class `cx_root`.

☑ The sequence of handlers is important. If the order of the two handlers were swapped, the system would use the handler for all exceptions of the class `cx_root` and the other handler would never be used. To avoid this, the compiler always checks that handlers are listed in the program in ascending order, with respect to inheritance.

☑ If a procedure is called within a `TRY` block, the appropriate handlers in the corresponding `TRY` ... `ENDTRY` construct will also catch all exceptions that are raised but not caught within that procedure.

If the system cannot find an appropriate handler in a `TRY` ... `ENDTRY` construct, it searches for handlers in the next outer `TRY` ... `ENDTRY` construct. This is the `TRY` construct whose `TRY` block was entered previously.[4] If it cannot find an appropriate handler there either, it continues the search. If it eventually finds a handler — which may be several steps upward in the call hierarchy — it stops processing all procedures, loops, etc., and goes immediately to the appropriate handler.

> ✓ ***Note!***
>
> *Only exceptions that occur in the TRY block can be caught by a handler of the same TRY ... ENDTRY construct. Since the TRY block comprises only the statements between TRY and the first CATCH, exceptions inside the handler coding cannot be caught by a handler in the same TRY ... ENDTRY construct. However, since TRY constructs may be arbitrarily nested, exceptions in handlers can be easily caught by local TRY constructs inside of the handler.*

When a handler catches an exception, the normal flow of control is changed. In many cases, ending procedures prematurely in this way causes objects to be left in an inconsistent state or prevents resources from being released. In particular, if the system jumps several levels up the hierarchy, the handler might not be able to take the appropriate corrective measures. For this reason, there is another clause for the `TRY` construct: the `CLEANUP` clause. It consists of the keyword `CLEANUP` and a number of statements that form the `CLEANUP` block.

[4] This also explains why, in the example, the first handler can also catch exceptions that are raised in method `m3`.

Listing 2: Using the CLEANUP Clause

```
 1 TRY.
 2     CALL METHOD o1->m1.    "Can raise an exception of the class cx_my1
 3     PERFORM f1.
 4   CATCH cx_root.
 5     "Handler for all exceptions
 6     "--- any number of statements ---
 7 ENDTRY.

 8 FORM f1 RAISING cx_my.
 9     TRY.
10       IF ... . RAISE EXCEPTION TYPE cx_my2. ENDIF.
11       CALL METHOD o1->m3.  "Can raise an exception of the class cx_my3
12     CATCH cx_my1 cx_my3 INTO ex.
13       RAISE EXCEPTION TYPE cx_my4.
14     CATCH cx_my4.
15       "Handler for exceptions of type cx_my4
16       "--- any number of statements ---
17     CLEANUP.
18       "CLEANUP block, to restore a consistent state
19       "--- any number of statements ---
20   ENDTRY.
21 ENDFORM.
```

The statements of the CLEANUP block are executed whenever an exception occurs in a TRY block that is not caught by the handler of the same TRY ... ENDTRY construct, but rather in a surrounding TRY construct. Within the CLEANUP clause, the system can, for example, restore an object to a consistent state or release external resources.[5]

Each TRY ... ENDTRY construct contains a maximum of one CLEANUP clause. You must include it after the last CATCH and before the ENDTRY, if at all. If there is a CLEANUP clause, the system executes the statements of the CLEANUP block and then propagates the exception upward. This propagation may cause the system to then execute the statements in other CLEANUP blocks (in surrounding TRY constructs) until the exception is finally caught by an appropriate handler.[6]

The CLEANUP clause exists solely to restore objects to a consistent state. Since there is already a handler for the current exception waiting to be executed, the clause must be left in the "normal" way through the ENDTRY. This means that any statement that would force the system to leave the CLEANUP block prematurely — such as RETURN or REJECT[7] — is forbidden. If an exception occurs inside the

[5] This is different from Java's `finally` clause, which is always executed. If you simulate Java's `finally` clause in ABAP, you have to put the statements from Java's `finally` clause in the CLEANUP clause *and* after the ENDTRY. In order to simulate ABAP's CLEANUP clause in Java, you have to introduce a flag that controls the execution of the statements in the `finally` block.

[6] The system processes CLEANUP clauses only if an exception is caught by a handler on its way through the call hierarchy. If there is no handler available, a runtime error occurs. In this case, the system does not process the CLEANUP clauses, so that it can access the relevant objects in the state they were in when the exception was raised.

[7] Statements such as EXIT, CHECK, or CONTINUE are forbidden only if they would really lead the system to quit the CLEANUP block, which, for example, is not the case if they appear in a loop *inside* of the CLEANUP clause.

Listing 3: The Syntax of the TRY … ENDTRY Construct

```
TRY.
    "--- Statements of the TRY block ---
  CATCH cx_a1 ... cx_an [INTO ex_a].
    " Handling exceptions cx_a1 to cx_an , n>=1
  ...
  CATCH cx_z1 ... cx_zn [INTO ex_z].
    " Handling exceptions cx_z1 to cx_zm , m>=1
[ CLEANUP.
    "--- Statements of the CLEANUP block ---
]
ENDTRY.
```

CLEANUP block, and is not dealt with inside this block, a runtime error occurs.

Listing 2 clarifies how the cleanup clause works: if in line 2 an exception of the class cx_my1 occurs, the system goes immediately to the handler in line 4. In line 3 there is a call to subroutine f1. If inside of f1 an exception of the class cx_my2 occurs (line 10), then there is no handler in the TRY ... ENDTRY construct of lines 9-20 available to catch it. However, since the handler in line 4 catches the exception, the system first executes the statements in the CLEANUP block (lines 17-19) and then goes immediately to the handler (line 4).

If in line 11 an exception of the class cx_my3 occurs, it is caught by the handler in lines 12-13 and stored in the variable ex. Since the handler is also used for exceptions of class cx_my1, the variable ex must have the type of a superclass of cx_my1 and cx_my3. The handler then raises an exception of the class cx_my4. Note that the handler in line 14 does not catch this new exception since it did not occur in the TRY block. For the same reason, the system does not execute the statements in the CLEANUP block, but instead goes immediately to the handler in line 4.

Listing 3 summarizes the basic syntax of the catching and handling exceptions. You can define as many handlers as you want (even none). The CLEANUP area is optional. Each CATCH clause can have one or more exception classes. The INTO addition is optional.

Declaring Exceptions

Exceptions that are not caught lead to a short dump. Thus, if you want to write a program that is not prone to regular short dumps, it is important to know the exceptions raised by the procedures[8] you call.

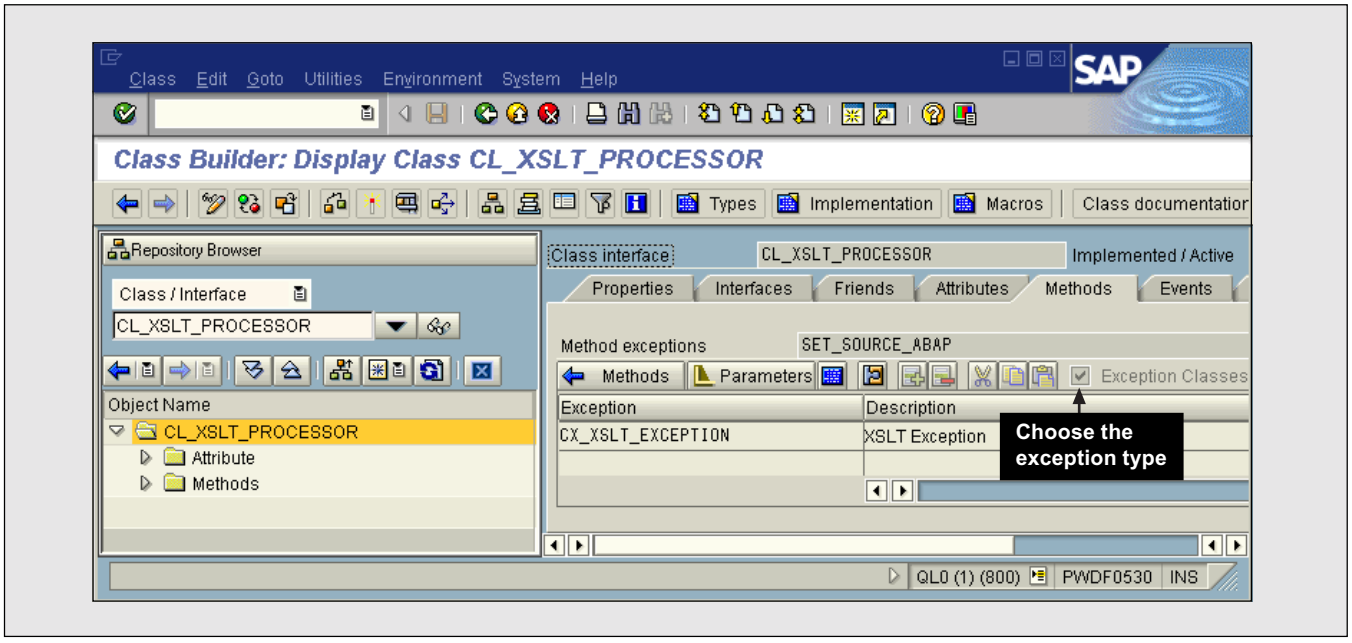
Therefore, all exceptions that may occur when a procedure is called must be declared in the procedure definition. Such a declaration must serve not only documentary purposes, but also as a guarantee for the procedure caller that no other exceptions are to be expected than those listed. The following sections explain when and how to declare class-based exceptions, and to what extent these declarations are used.

Syntax and Semantics of the Declaration

You declare an exception by using the RAISING clause when defining a procedure. After the RAISING keyword, you must list the classes of all

[8] The term *procedure* is the generic term for a method, function module, or subroutine.

Figure 1 Declaring Exceptions in the Class Builder



the exceptions that the procedure may raise. In the Class Builder or Function Builder, select the exception button for a procedure, select the checkbox for class-based exceptions, and list the exception classes in the table control (see **Figure 1**).

Note that each class declared in the `RAISING` clause comprises all of its subclasses as well.[9]

To provide the guarantee for the procedure caller that only the exceptions listed in the `RAISING` clause leave the procedure, the compiler checks that all exceptions that could be raised, but are not caught locally, are declared in the interface.[10] In order to support changes in the procedure's interface, the compiler displays a warning in case of a missing exception declaration — it does not stop the compilation with a syntax error.

The reason for issuing a warning instead of an error is that in real systems, procedure interfaces do in fact change, and in huge application systems like the SAP application server, there can be many applications that use a particular procedure. Obviously you cannot invalidate all those applications simply because they reference such a procedure for some rarely used feature. Therefore, by issuing just a warning, an application can be generated even if exceptions have been added to the procedure interface used by the application.

Since the compiler issues only a warning instead of an error, and since the programmer could have simply ignored the warning, the compiler has *not* provided the guarantee to the procedure caller that only exceptions listed in the `RAISING` clause may leave the procedure. The guarantee must therefore be provided by the runtime system: if an exception leaves the call interface of a procedure without having been declared in a `RAISING` clause for that interface, a runtime error occurs.[11] This is much better than enforcing source code changes in all clients, which would mean that the system cannot be

[9] If you want to emphasize that an exception of an implicitly declared subclass can be raised by a procedure, you can explicitly list the subclass in the `RAISING` clause, too.

[10] For consistency, the compiler checks only method interfaces. To check function modules and subroutines, use the extended program check (transaction SLIN).

[11] This error situation raises an exception in turn. See the upcoming section "Violation of the Interface" for more information.

used before all changes have been completed. So, the system keeps running as long as the erroneous situation doesn't occur.

In summary, the RAISING clause fulfills two objectives:

1. It is used by the runtime system to provide the guarantee for the procedure caller that it need only expect exceptions that are declared in the interface.

2. It lets the compiler confirm that all possible exceptions have been taken into account — whether they are to be caught locally or have been declared in the RAISING clause.

Drawbacks of an Enforced Declaration

There are errors that can occur almost everywhere, but which most developers do not want to or cannot, for some reason, deal with — such as resource bottlenecks or configuration errors. If we were to consistently apply the rules on declaring exceptions, we would have to declare such errors in almost every interface. This would *not* lead to more robust programs. It would simply make programs unreadable, less useful, or worse, *less* robust.

The reason for this surprising, counterproductive effect is that users who use procedures or statements in which such errors are raised have only the following three choices, none of which is satisfactory:

1. They must include the exception in their RAISING clause. Since we are talking about errors that can occur almost everywhere, and there are probably many such errors, this means that the RAISING clause of almost every procedure is cluttered with useless information.

2. They can catch the exception and terminate the program. This prevents a suitable handler far up in the call hierarchy from reacting to the exception.

3. If users think that the exception will not occur in their application, they write empty exception handlers — i.e., handlers that will catch the exception but do nothing. This is the worst possible choice since it prevents the detection of bugs if the exception *does* occur. Unfortunately, it is probably also the choice that most users would make.

To get around this problem, there is a class called cx_no_check whose subclasses do not need to follow the usual declaration rules[12]: neither the compiler nor the runtime system will check whether exceptions derived from cx_no_check are listed in the RAISING clause. The user of a procedure must be aware that these exceptions may be raised in addition to those listed in the RAISING clause.

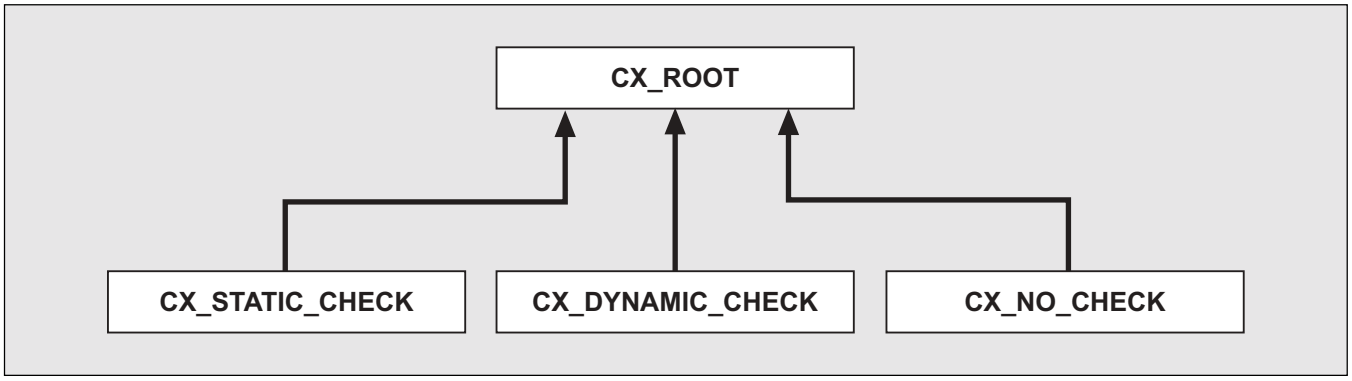
There are also errors for which you want to provide the guarantee that they leave the procedure only if they are listed in the RAISING clause, but you do not want to force users of that procedure to deal with an occurrence of the error. These exceptions, such as cx_sy_zerodivide, are raised if a procedure is called with wrong parameter values (such as a zero divisor for the operation "/") or, more generally, if its precondition is violated.

Users of procedures or statements that can raise such an exception do not want to be forced by the compiler to either catch the exception or include it in their own RAISING clause. If you divide by 2, you *know* that cx_sy_zerodivide will not occur, although it can in general occur during a division. Thus you do not want the compiler to check whether you have dealt with the exception.

Does this also mean that you do not want the *runtime system* to check the RAISING clause, as with exceptions inheriting from cx_no_check? The answer is no. To see why, let us assume that cx_sy_zerodivide is derived from cx_no_check, and that you write a procedure where you invert some number (i.e., divide 1 by that

[12] Exception classes derived from cx_no_check cannot be declared in a RAISING clause.

Figure 2 The Top Classes of the Exception Hierarchy



number) that you *thought* would never be zero. If you were wrong, and an exception of class `cx_sy_zerodivide` is raised, then you have clearly introduced a programming bug. There are two possible outcomes:

1. There is no handler for this exception of class `cx_sy_zerodivide` and a short dump occurs. The short dump shows the position of the division operation inside the procedure and also shows the call hierarchy, but there is no clear hint that *your* procedure was the culprit.

2. There is a handler for exceptions of class `cx_sy_zerodivide` far up in the call hierarchy that will catch *all* exceptions of type `cx_sy_zerodivide`, even the one that was induced by the programming bug. This would prevent the detection of such errors in your coding.

While the first scenario is annoying, the second scenario is dangerous. To avoid these scenarios, `cx_sy_zerodivide` is not derived from `cx_no_check`, but from a special class `cx_dynamic_check`, which has the property that its subclasses are checked only by the runtime system to determine whether they are listed in the `RAISING` clause. Consequently, the runtime system will now check the `RAISING` clause of your procedure. Since `cx_sy_zerodivide` is not listed there, the exception induced by your programming bug will lead to a short dump. It includes a message indicating that the short dump was induced by your procedure, because it neither catches the exception locally nor propagates the exception.

In summary, the benefit of this approach is that exceptions derived from `cx_dynamic_check` do not annoy users who know that these exceptions cannot occur (as would be the case with exceptions derived from `cx_static_check`), and the problem is not shifted to others if these "impossible" exceptions *do* in fact occur (as would be the case with exceptions derived from `cx_no_check`).

Categories of Exceptions

The top classes in the exception hierarchy look like **Figure 2**.

All four classes (`cx_root`, `cx_static_check`, `cx_dynamic_check`, and `cx_no_check`) are abstract. The main properties of the three subclasses directly below superclass `cx_root` are as follows:

- A `cx_static_check` exception can leave a procedure only if it has been declared in the `RAISING` clause of the procedure's interface. As mentioned in the previous section, both the compiler and the runtime system will check that all the exceptions from this category, that may occur but that are not handled locally inside of the procedure, have been declared.

- A `cx_dynamic_check` exception can leave a procedure only if it has been declared in the `RAISING` clause of the procedure's interface. Only the runtime system performs this check once an exception tries to leave a procedure.

- A `cx_no_check` exception can *always* leave the interface of a procedure. Neither the compiler nor the runtime system performs any interface checks.

If you define a new exception class for your application, *choosing the right category* is the most important decision:

- The `cx_static_check` category should be chosen if you want to make sure that this exception is always dealt with and if a local exception handler has a chance to do something useful in the exceptional situation. If you choose `cx_dynamic_check` when `cx_static_check` should have been chosen, you lose the assurance from the compiler that the exception is always dealt with — i.e., short dumps might occur, indicating that the exception has not been dealt with. If you choose `cx_no_check` instead, you make programming in your application more difficult since this exception must now be considered ubiquitous; if the exception leads to a short dump, it is not easy to see who is responsible for it.

- The `cx_dynamic_check` category is similar to the `cx_static_check` category and should be chosen if programmers can usually avoid the occurrence of the exception beforehand; in most cases the programmers know that the exception will not be raised in their application context. If you choose `cx_no_check` when you should have chosen `cx_dynamic_check`, you have the same problem as before (i.e., you lose the assurance that the exception is dealt with). If you choose `cx_static_check` instead, you risk making programming in your application clumsy, with the consequence that people write empty exception handlers that prevent the detection of programming bugs.

- The category `cx_no_check` should be chosen if the exception can occur almost everywhere, but most programmers do not want to or cannot deal with such an exception, such as resource bottlenecks. If you choose `cx_static_check` or `cx_dynamic_check` when you should have chosen `cx_no_check`, users are likely to write empty exception handlers,[13] which make it hard to detect bugs.

Violation of the Interface

If the interface is violated by an exception of the category `cx_static_check` or `cx_dynamic_check`, the program does not terminate with a runtime error. Instead, an exception of the class `cx_sy_no_handler` is raised and a reference to the original exception is stored in the exception. Any handler for an exception of the class `cx_sy_no_handler` catches a program error from the called procedure, not the original exception. You should really think twice before catching such an exception,[14] but there are situations where it could be useful to do so (e.g., for a temporary bug workaround).

Exceptions in Event Handlers and Class Constructors

There is no `RAISING` clause defined for event-handler methods. This is consistent since the concept of "expecting" specific exceptions violates the

[13] If you are in a situation where you get a `cx_static_check` or `cx_dynamic_check` exception that should have been declared as `cx_no_check`, do not write an empty exception handler. Throw your own exception of category `cx_no_check` instead.

[14] Remember that a suitable exception handler is found if the class of the exception, or one of its superclasses, is listed in the corresponding `CATCH` clause. Thus any handler that looks like `CATCH cx_sy_no_handler`, `CATCH cx_no_check`, or `CATCH cx_root` must also be prepared to handle this kind of exception. A handler should at least write an error log in this case.

principle of the event mechanism. The trigger of an event does not make any assumptions about potential event handlers that are registered for the event. An event is a one-way street — the trigger provides information to the event handler (or handlers), but it does not expect anything from them.

Thus all `cx_static_check` and `cx_dynamic_check` exceptions that occur in event handlers, but are not caught there, cause an interface violation. An exception of the class `cx_sy_no_handler` raised in this way can be caught beyond event boundaries, just like other exceptions of the category `cx_no_check`.

Similarly, you cannot define a `RAISING` clause for class constructors. The user of a class generally does not know that it is using the class for the first time and thus that the class constructor is executed.

Defining Exceptions

Exception classes can be global or local classes. When creating a global exception class, the Class Builder ensures that its name follows the `cx_` prefix naming convention.[15] Exceptions have some specific properties — e.g., each exception has an explanatory text assigned to it that describes the exception. In the following subsections, we look at these properties and see how the Class Builder has been adapted to support them. If you define local exception classes, you will of course have no such tool support, so you have to take care of these aspects yourself.

Constructors for Exceptions

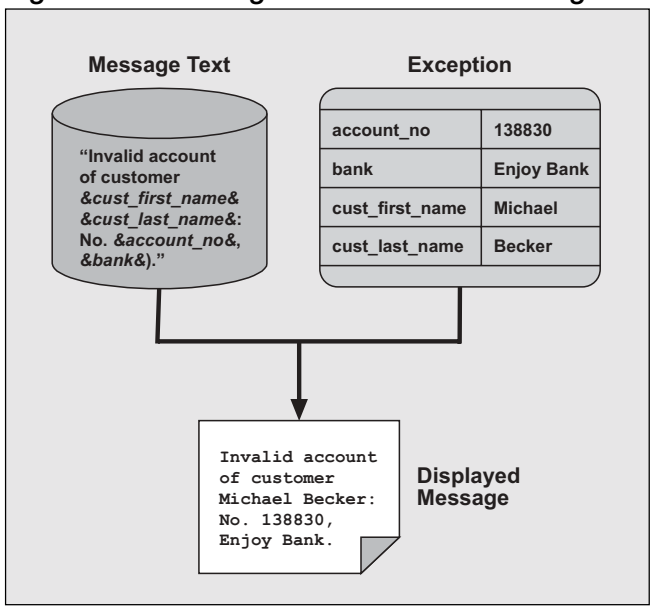
Exceptions are usually created and raised immediately. After calling the constructor of an exception, you usually do not invoke a sequence of set methods or attribute assignments. That means that all attributes are usually set via the constructor. Following this observation, the exception constructor is automatically created as soon as you save the exception class. It has one optional parameter for each non-private attribute.

Texts for Exceptions

Each exception has an explanatory text assigned to it that describes the exception. The text is used for error analysis in error logs or short dumps and therefore describes the exception from a technical point of view. A good user interface therefore usually catches exceptions and describes the error from a user point of view, instead of simply forwarding the technical message from the exception object.

For texts to adequately describe an exception, you have to be able to set parameters for them. **Figure 3** shows an example.

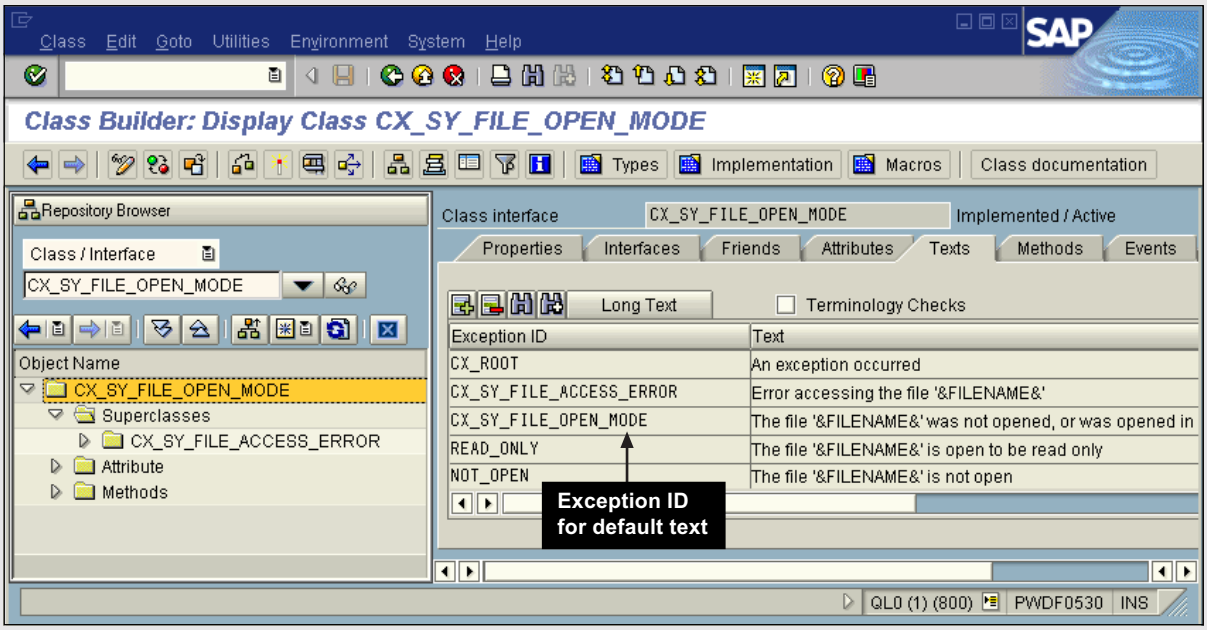
Figure 3 Setting Parameters for Messages



In this example, it is not only important to display that the bank details are incorrect, but also to display the name of the account holder. For this purpose, you can set parameters for exception texts using (textual) attributes of the exception object. In this context, *textual* means that the attribute has a scalar type (for example, it is a C field, a string, an

[15] With the usual customer namespaces the prefixes `ycx_`, `zcx_`, and `/cust/cx_` are valid as well.

Figure 4 An Example Text Assignment



integer, etc.). This rules out internal tables, object references, and structures.

Attributes can be included in the text by using the name of the attribute surrounded by "&," as in "`&name_of_attribute&`." If you want the "&" symbol to appear in the actual text, you have to enter it twice.

Consider the following example:

```
'Law Firm of &lawyer&&&&partner&'
```

together with the following attributes:

```
lawyer   = 'Smith'
partner  = 'Brown'
```

produces the following text:

```
'Law Firm of Smith&Brown'
```

You might want to define more than one text for one exception class if you have two almost identical error situations. For those kinds of exceptions, most users do not want to define an exception class for each of the two error situations, but it might be useful to provide separate texts for an error analysis of the two error situations.

The texts are stored in the Online Text Repository (OTR). The exception object contains only a key that identifies the text (together with the system language). To make it easy to define texts for an exception, the Class Builder provides an additional "Texts" tab for exception classes. For each text you insert there, you also specify an *exception ID*, which is a language-independent handle to the text. For the *default text*, the exception ID always has the same name as the name of the exception class. **Figure 4** shows an example.

If you raise an exception without specifying a text, the default text will be selected automatically. You can create an exception using another text if you provide a value for the parameter `textid` of the exception constructor. Each exception ID

corresponds to a class constant that holds the OTR key for the corresponding text. Thus, to create and raise an exception `cx_sy_file_open_mode` using the last text in Figure 4, use:

```
RAISE EXCEPTION TYPE
    cx_sy_file_open_mode
  EXPORTING textid =
   cx_sy_file_open_mode=>read_only.
```

Attributes and Methods of Exceptions

Attributes of exceptions are used to give the handler more information about the error situation. You can define attributes for exception classes like you would for normal classes.

Exception classes inherit the following *attributes* from `cx_root`:

- `textid` — Used to define different texts for exceptions of a particular class, as explained above. Affects the result of the method `get_text` (see below).
- `previous` — If one exception is mapped to another, this attribute can store the original exception, which allows you to build a chain of exceptions. If a runtime error occurs, the short dump contains the texts belonging to all the exceptions in the chain. Mapping one exception to another may be advisable if, for example, the context in which the first exception occurred is important for characterizing the error situation represented by the second exception (see the upcoming section "Mixed Use of Class-Based and Non-Class-Based Exceptions").

Exception classes inherit the following *methods* from `cx_root`:

- `get_text` — Returns the textual representation as a string, according to the system language of the exception, as explained earlier in the section "Texts for Exceptions" (see also the attribute `textid` above).
- `get_longtext` (available only since 6.20) — Returns the long variant of the textual representation of the exception as a string (see also the attribute `textid` above). In the Class Builder, you can also specify a *long text* for each exception ID.
- `get_source_position` — Returns the program name, include name, and line number reached where the exception was raised.

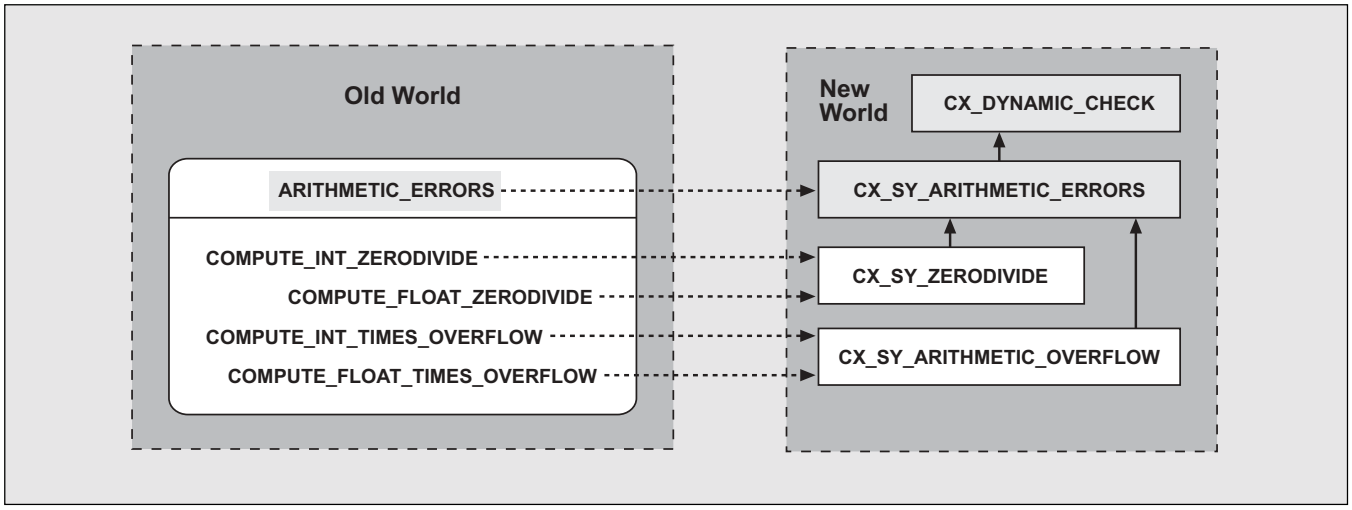
Apart from the constructor, which is automatically constructed, as explained earlier in the section "Constructors for Exceptions," no methods can be defined in Release 6.10. Neither can you specify aliases, interfaces, friends, events, and internal types for exception classes. This restriction has been lifted with Release 6.20.

Interaction with Existing Error-Handling Methods

As described at the beginning of the article, there are already several types of error handling available in ABAP, namely:

- The classical ABAP exceptions that are raised by `RAISE` and that can be handled by calling methods and function modules using the `EXCEPTIONS` clause. These exceptions cannot be handled when calling subroutines; instead they are propagated automatically to the next non-subroutine level.
- Some of the runtime errors that can be caught (i.e., *catchable* runtime errors) if the statement that caused the runtime error is placed in a `CATCH SYSTEM-EXCEPTIONS` block. Note that none of the runtime errors is propagated across modularization entities.
- Messages in the `MESSAGE` statement that can also indicate an error. If the message is of type "E" or "A," it can be caught directly by calling methods and function modules using the `EXCEPTIONS` clause and the special (classical) exception name `error_message`.

Figure 5 Assignment of Runtime Errors to Exceptions



All these error-handling methods are subsumed by class-based exception handling. However, the concepts behind these methods are too different to define a suitable mapping to class-based exceptions. For this reason, these next sections explain how the concepts interact.

Classical Exceptions in Methods and Function Modules

Classical exceptions and the newer class-based exceptions are two totally different concepts, so there is a strict separation between them. In other words, classical exceptions cannot be caught by the `CATCH` clause of a `TRY` block. Conversely, a class-based exception cannot be caught via an `EXCEPTIONS` clause.

To drive the separation even further, each procedure has to decide if it wants to raise *either* the old *or* the new exceptions. It's syntactically impossible to specify a `RAISING` clause and an `EXCEPTION` clause at the same time for the same procedure. If you decide in favor of the new exceptions, you cannot use any `RAISE` (or `MESSAGE ... RAISING`) statement, and if you opt for the old ones, you cannot use the new `RAISE EXCEPTION`.

Of course, in the implementation of your procedure you might have to call other procedures that may raise the other kind of exception. If you can't handle these exceptions locally, you will have to map them — i.e., catch the exception and raise another exception of a proper type.

Migration of Catchable Runtime Errors

The process of catching runtime errors using `CATCH SYSTEM-EXCEPTIONS` is now obsolete. A number of rules apply to ensure a simple model for coexistence between the old and new ways.

For every runtime error that can be caught by `CATCH SYSTEM-EXCEPTIONS`, there is now a corresponding exception class that can be used in a `TRY ... ENDTRY` construct. Grouping of runtime errors is also supported by respective classes and the use of inheritance. Note that some runtime errors that are virtually the same are now mapped to the same class. For example, `compute_int_zerodivide` and `compute_float_zerodivide` are both mapped to class `cx_sy_zerodivide`[16] and `arithmetic_errors` (the group of all arithmetic errors) is now represented by class `cx_sy_arithmetic_errors` (see **Figure 5**).

[16] If you really need to differentiate between the two types of division by zero, you can do so by inspecting the exception object.

Although CATCH SYSTEM-EXCEPTIONS is now obsolete, it can still be used for backward-compatibility reasons. Note, however, that any new catchable runtime errors will only be catchable via TRY … ENDTRY using the proper exception class, but not via CATCH SYSTEM-EXCEPTIONS.[17]

For coexistence between the old and new ways, the following rules apply:

- ☑ You cannot use CATCH SYSTEM-EXCEPTIONS and TRY constructs or the statement RAISE EXCEPTION in *one* procedure simultaneously. If a procedure interface contains a RAISING clause, you cannot use CATCH SYSTEM-EXCEPTIONS in this procedure.

- ☑ If a catchable runtime error occurs in a procedure without CATCH SYSTEM-EXCEPTIONS, a class-based exception is raised. The rules for class-based exception handling mentioned in the previous section apply here.

- ☑ If a catchable runtime error occurs in a procedure that uses CATCH SYSTEM-EXCEPTIONS, the runtime system tries to catch the error according to the semantics of CATCH SYSTEM-EXCEPTIONS. If this fails, the new class-based exception handling comes into play: since there is no local handler, the exception handling tries to directly pass the local interface and then looks for a suitable handler in the call hierarchy. If there is no handler available, a runtime error occurs.

Messages in the MESSAGE Statement

The MESSAGE statement in ABAP has two functions for most message types:

- Indicating an error and ending the normal program flow
- Selecting an error handler according to the message type (and context); all handlers display the message text

These functions are strictly separated from each other in class-based exception handling. In particular, no assumption is made about possible handlers at the place where an exception is raised. When you raise an exception, the error has to be characterized in just enough detail (by selecting the correct exception class, for example) that the correct handler is activated.

Although message texts are not primarily intended for interaction with end users, it can sometimes be useful to display them using the current message concept.

In order to display the exception object's exception text, you can use the MESSAGE statement in the following way: first retrieve the textual representation by a call of get_text, then pass this text — together with a suitable message type — along to the MESSAGE statement. The example in the upcoming section "Creating a History of Exceptions" demonstrates this. In Release 6.20 you will be able to pass the exception object directly to the MESSAGE statement.

Restrictions

Not all system exceptions can be caught in all contexts, which was already the case for the old system exceptions (CATCH SYSTEM-EXCEPTIONS). For example, you can catch conversion errors in MOVE, but not in SELECT. Although there are plans to increase the number of runtime errors that can be caught, and the number of contexts in which they can be caught, this will be an evolving process.

Exceptions in a class constructor or a conversion exit that are not handled locally lead to a runtime error. This is comparable to exceptions occurring in CLEANUP clauses. The reason is simple: since both run behind the scenes, users cannot be aware of these exceptions, so they must be handled locally.

[17] For example, since Release 6.10, many SQL errors can be caught using the new exception mechanism, but not using CATCH SYSTEM-EXCEPTIONS.

Listing 4: Set the "Previous" Attribute in the Handler Code

```
TRY.
  ...
CATCH cx_low_level_cxception INTO caught_exception.
  RAISE EXCEPTION TYPE cx_higher_level_exception
        EXPORTING attr1 = ...
                  previous = caught_exception.
ENDTRY.
```

Common Uses of Exception Handling

To familiarize you with the new exception-handling concept, in the following sections we provide you with step-by-step descriptions of some common use cases.

Return Codes Versus Exceptions

Before we go into detail on how to use the exceptions, we should say a little about when it is appropriate to use exceptions at all. We often notice that programmers use exceptions when it would be better to use return codes.

As a rule of thumb, you should always ask yourself if a certain situation is really exceptional or if a user can expect the situation as a "normal" event. For example, suppose you write a general search routine that looks for a specific element in a collection. If the element is not found in the collection, which is generally a "normal" event, it is better to use a special return code to indicate this rather than to go so far as to raise an exception.

On the other hand, there are situations that are clearly exceptional, as in the division-by-zero example given earlier, where raising an exception is the proper way to deal with the situation.

For the majority of situations however, it depends on the perception that you have. For example, let's assume that you want to define a method to delete some element from a collection. How should you behave in situations where the element you'd like to delete doesn't exist? The answer is that you will have to decide whether the situation is exceptional or not. If the element is expected to exist, you should raise an exception; if not, just use a return code (if at all).

Creating a History of Exceptions

Suppose you want to handle an exception by raising a new exception that is more meaningful to the user in the specific situation. Nevertheless, you still want the original exception to be included in the history of the new exception.

Every exception has a `previous` attribute. To create a history of exceptions, this attribute has to be set in the handler code, as shown in **Listing 4**.

In the event that there is no handler that catches the new exception, a short dump will occur. In the short dump you will see the complete history of all exceptions that are linked together by the `previous` attribute.

Displaying Exception Messages on the Screen

Let's assume that you want to display the text of a caught exception on the screen, as was previously possible for "T100" messages.

Listing 5: Output the Text Using MESSAGE

```
DATA str TYPE string.
  ...
TRY.
  ...
CATCH cx_some_exception INTO caught_exception.
  str = caught_exception->get_text(  ).
  MESSAGE str TYPE 'E'.
ENDTRY.
```

The MESSAGE statement has been extended, making it possible to also output strings. The handler simply has to get the text, and it can then output the text using MESSAGE, as shown in **Listing 5**.

Although exceptions can be easily mapped to messages, note that exception texts are primarily intended as a way to describe an exception if it is not caught and it ends the program. Interaction with the user should be separate from the exception texts and handled independently.

Mixed Use of Class-Based and Non-Class-Based Exceptions

First, assume you are in the process of writing a method m_new and you want to use class-based exceptions in the interface. In the implementation, you use a function module f_old that raises old exceptions and you want to propagate that exception to the caller of m_new.

Since you cannot use old and new exceptions in combination, you will first have to define an exception class for the exception you want to propagate, such as cx_m_new. The coding then looks like **Listing 6**.

Now let's look at the opposite scenario: suppose you already have a function module f_old with the interface containing an EXCEPTIONS clause. Assume you want to call method m_new, which raises new exceptions, in the implementation of f_old. Since you cannot propagate the new exception, you have to either handle it or map it onto an old exception. In both cases, you must catch the new exception using a TRY construct in f_old, as shown in **Listing 7**.

If the function module f_old does not yet have an EXCEPTIONS clause, and the exception from m_new is not to be handled locally, this exception can be propagated simply by declaring it in a RAISING clause that belongs to f_old.

Adding Exceptions to a Method That Is Already in Use

Assume that a method do_something may raise exceptions of the class cx_something (declared in the RAISING clause of do_something) and that this method is already used by many other procedures. As part of the maintenance procedure, the method is to be modified so that it can raise a new exception called cx_new.

Listing 6: Define a Class to Propagate the Exception

```
METHODS m_new RAISING cx_m_new.
....
METHOD m_new.
  ...
  CALL FUNCTION 'F_OLD' EXCEPTIONS not_found = 1.
  IF sy-subrc = 1.
*    -- Mapping old exception onto new exception
    RAISE EXCEPTION TYPE cx_m_new.
  ...
ENDMETHOD.
```

Listing 7: Catch the New Exception Using a TRY Construct

```
FUNCTION f_old EXCEPTIONS old_exc_1 ... old_exc_n.
  ...
  TRY.
    CALL METHOD o->m_new.
  CATCH cx_m_new.
*   -- Mapping new exception onto old exception
    RAISE old_exc_1.
  ENDTRY.
  ...
ENDFUNCTION.
```

There are two different possibilities here:

a. `cx_new` is a subclass of `cx_something`. In this case, the interface of the method `do_something` has not changed. The `RAISING` clause is still valid and the procedures that use it do not need to be adjusted.

b. `cx_new` is *not* a subclass of `cx_something`. This is an interface modification, since the caller of `do_something` can get a new type of exception that was not expected until now. In this case, the developer has to add the new exception class to the `RAISING` clause:

```
METHOD do_something RAISING
  cx_something cx_new.
```

As a result, the previous procedures that used `do_something` may be invalidated since they did not expect the new exception `cx_new`; they did not catch this exception locally or specify it in their interface.

To simplify the modification process in these cases, invalidation does not cause a syntax error, as explained earlier in the section "Syntax and Semantics of the Declaration."

Conclusion

In these pages we have introduced a unifying concept for all kinds of errors and other exceptional situations. Such a concept is essential for writing generic services that are to be used in various contexts.

This article has demonstrated the advantages of this concept and hopefully has enabled you to use exceptions in new and existing programs. To summarize the advantages:

- ☑ There is a clear separation between error detection and error handling, and a clear separation between "normal" coding and error handling. An error induces a change of control to the appropriate error handler.
- ☑ The use of classes makes it possible to support exception handlers, for specific exceptions and situations where you need to know every detail, and generic handlers, which are suitable for a whole group of exceptions.
- ☑ The automatic exception propagation across all kinds of procedures makes exception handling usable without forcing you to use a specific way to modularize your programs.
- ☑ The concept of "checked" exceptions enables you to find untreated exceptions directly at compile time or makes it possible to identify the procedure that should have handled the exception. The different levels of checks avoid counterproductive "check workarounds" like empty exception handlers, which would sometimes be necessary with a less flexible concept.
- ☑ Programmers have the ability to specify some "cleanup" coding that will be executed only if a procedure is left prematurely.

Although using new exceptions does not force you to abandon existing procedures that use classical exceptions, once you discover how easy the new exceptions are to use, and the enhanced functionality they provide, you will prefer them anyway — just give the `TRY` a try!

Gerd Kluger studied computer science at the University of Kaiserslautern, Germany. After receiving his degree, he worked for a company whose main focus was the development of programming languages for business applications. He was responsible for the development of the compiler and programming environment for the object-oriented programming language Eiffel.

Gerd joined SAP AG in 1998 and since then has been working in the Business Programming Languages Group. His main responsibility is in the development of ABAP Objects and the further development of system interfaces, especially with regards to the file system.

He can be reached at gerd.kluger@sap.com.

Christoph Wedler received a degree in computer science from the University of Erlangen, Germany, in 1993. He joined SAP in 1999 and is now a member of the Business Programming Languages Group. Christoph is responsible for the integration of XML into the ABAP language and different parts of the ABAP runtime environment. He can be reached at christoph.wedler@sap.com.

