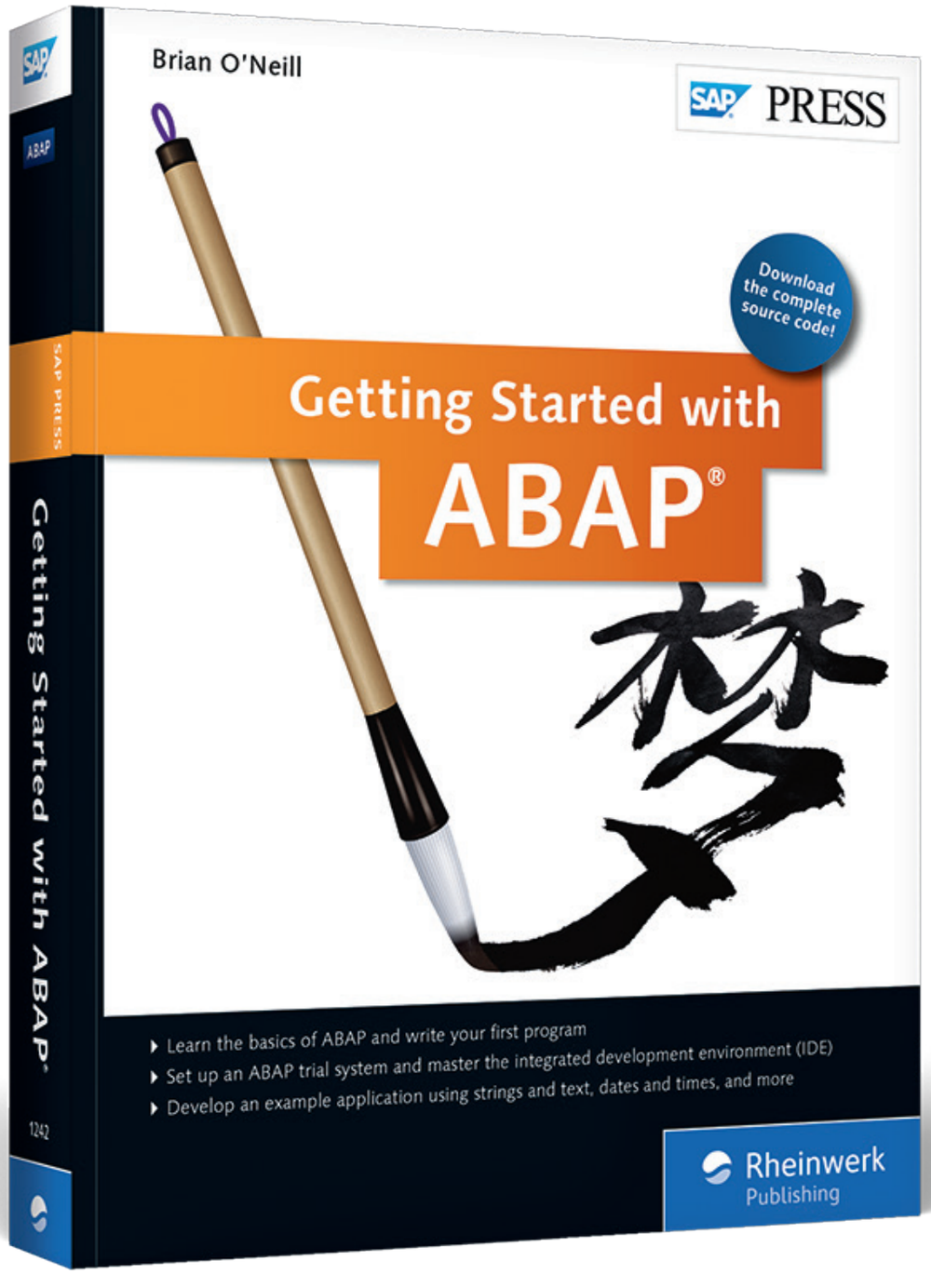


Reading Sample

Modularization involves placing specific sequences of ABAP statements in a module, instead of placing all the statements in a single main program. In this sample chapter, you will learn to modularize your program using object-oriented programming, with classes, global classes, function modules and subroutines.

Brian O'Neill

Getting Started with ABAP

451 Pages, 2016, $49.95/€49.95
ISBN 978-1-4932-1242-2

www.sap-press.com/3869

Making Programs Modular

6

So far, this book has covered a number of technical methods that allow you to do specific things using ABAP programs. You discovered different ways to process data, read data from a database, and work with data in working memory. In this chapter, we will discuss how to organize, or *modularize*, your program using object-oriented programming to complete all of these tasks. We will cover modularization using classes, global classes, function modules and subroutines.

What is modularization?

Modularization involves placing specific sequences of ABAP statements in a module, instead of placing all the statements in a single main program. There are two very important reasons that you need to modularize your program instead of having just one long program that executes from the beginning to the end. First, you want to write programs that are easy for other programmers to read and understand. Second, you want to be able to reuse common functions multiple times in a single program or across multiple programs and avoid redundancy.

In addition, modularization has the added benefit of making ABAP programs easier to support and enhance after they have been written.

Separation of Concerns

Separation of concerns is a principal used to separate a program into different layers, each with its own function. Imagine an ABAP program that was created to report on some data from the database. You could break that program into three different parts, one part to read the data from the database, another part to process the data, and a third part to display the results, as shown conceptually in Figure 6.1.

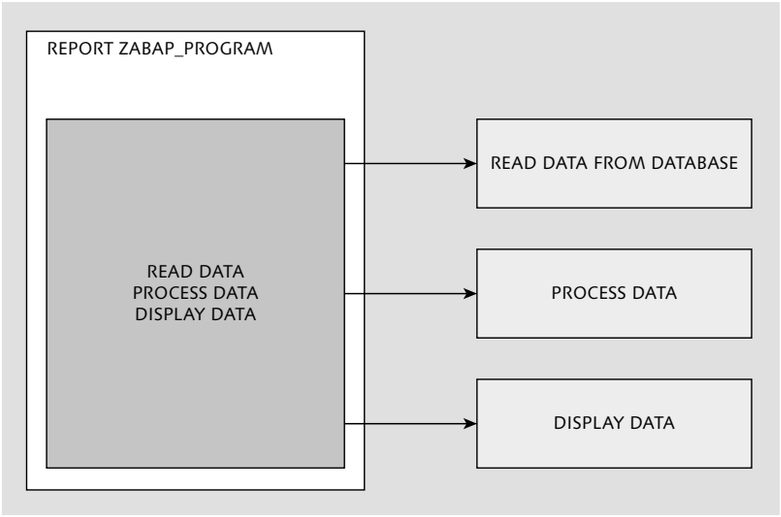


Figure 6.1 Using Seperation of Concerns to Break a Program into Single-Function Units

When you break your program into these three different sections, you then have one place to make changes to any of those functions.

Procedural programs

Back in the ancient days of computing, people would write long programs using punch cards that had to be executed from the beginning to the end and probably scheduled to run at a certain time. Today, applications are used by people in real time, which means that you need to change your application to meet the user's sometimes crazy expectations. Programs designed to run from the beginning to end are called *procedural programs*.

In a typical ABAP journey, it's normal to see an old program written in a procedural format—and then you'll hear from a user that the program is supposed to process data in some way that isn't working. You'll have to go through the long process of reading the program and debugging to figure out where the data is changed, only to find that the data is read and changed all over the place.

Why use separation of concerns?

In order to avoid writing procedural nightmare programs, use the separation of concerns principal to keep each unit focused on performing only one function, and name each unit based on the function that it performs. This makes it much easier to understand the program, fix it, and enhance it. Remember that you may write the program once, but someone else may have to fix or change it, possibly years later! Therefore, after using the plan in Figure 6.1, if users returned to you and said that they wanted additional data from the database, you would know exactly what unit to change, and if they wanted the ability to refresh the data, you would know that you can add the ability to call the read data from the database after displaying the data. If you had just one long program, it would be harder to find out exactly where you need to make changes, and you definitely would not be able to reuse any particular unit; the program would all be one long unit.

Of course, each person's interpretation of a unit focused on performing only one function might be different. That's where this concept can become more of an art than a science. If the units are made too small, it can be confusing because there are so many; if they are made too large, it can be confusing because there's so much code in a single unit. Remember that you're both writing a program and trying to make it easy for the next person to fix and enhance.

Naming different units

Figure 6.2 expands on the original conceptual drawing in Figure 6.1 to demonstrate breaking up and naming units for each function that they perform. This example demonstrates a program that gets a list of possible flights based on a search, calculates the price of the flight options, and displays the results to the user. Each unit completes one function, and each unit has a descriptive name. If a person said that there was an issue with the price being calculated, you would know exactly what unit of code he or she was talking about.

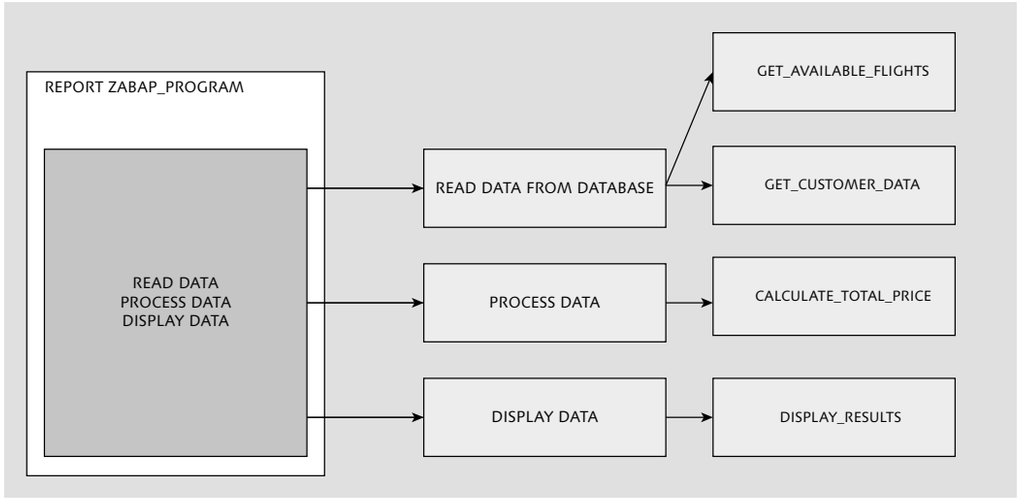


Figure 6.2 Seperation of Concerns with Named Code Units

Changing units of code

Just because you created certain units of code when the application was created doesn't mean that you can't add more. It's common to have to add additional functionality in a single unit of code, in which case you should determine if the additional functionality might need to be in its own functional unit of code. If each unit is expected to perform one function, ask yourself if the new code is really completing that same function or if it's its own unit. Also, anytime the new code is something that could be reused, then it should be in its own unit. Once your code is completed and working, it is always good practice to go back and see what kind of improvements you can make to your code and find any code that is repeated and could be modularized.

Now that you understand the concept of Separation of Concerns, we will cover how to utilize it using object-oriented programming.

Introduction to Object-Oriented Programming

OOP

The recommended method for modularizing ABAP programs is to use object-oriented programming. There are some people in the ABAP community who are unsure about object-oriented programming and have even posed the idea that there is ABAP and OO-ABAP (object-

oriented ABAP). The fact is that there is no ABAP versus OO-ABAP: just ABAP with good developers and bad developers.

If you have written object-oriented programs in other languages, you will find that there are a few ABAP quirks, but all of the concepts that you have seen in other languages will apply in ABAP as well.

What Is an Object?

A programming *object* is designed around the idea of objects in the real world. A good introductory conceptual example is that of a car. A car has attributes that describe its current state, such as fuel level and current speed, and attributes that describe the object, such as manufacturer and model. There are also a few methods that describe how we interact with the car, such as accelerate, decelerate, and refuel. Figure 6.3 shows a conceptual drawing of this car object.

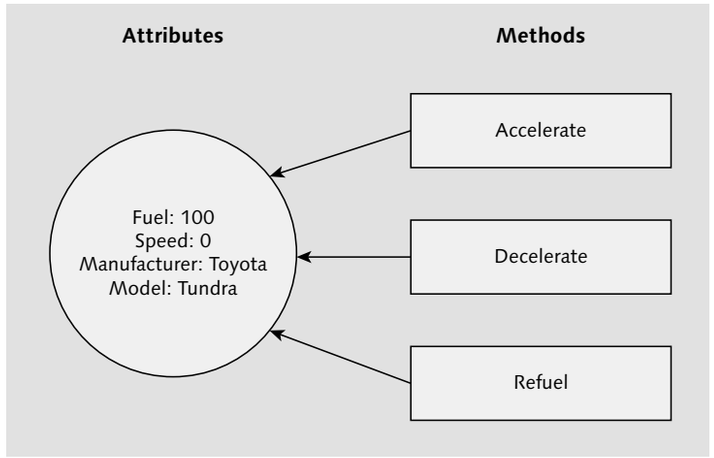


Figure 6.3 The Car Object

Each of the object's methods is a functional unit designed to complete one task. Each attribute of the object is a variable that all of the methods have access to. The pseudocode in Listing 6.1 shows what the code in the accelerate method could look like.

```
METHOD ACCELERATE.
     SPEED = SPEED + 5.
ENDMETHOD.
```

Listing 6.1 Pseudocode of Accelerate Method

Classes

Now, say that the example in Figure 6.3 specifically refers to a Toyota Tundra, but you want to create additional objects for different types of cars. This is where classes come in. Each object is actually an instantiation of a *class*, and you can have multiple objects of the same class. Again think back to the real-life example of a car; the Toyota Tundra is a type of car, and all cars can accelerate, decelerate, and refuel, but this particular car is going at a certain speed and has a certain fuel level. When creating objects, all of the code is stored in the class, but you can create multiple objects that use that code, and each will have its own set of attributes to describe it, as shown conceptually in Figure 6.4. You can think of the class in this example as a mold, whereas the objects are those items created from that mold.

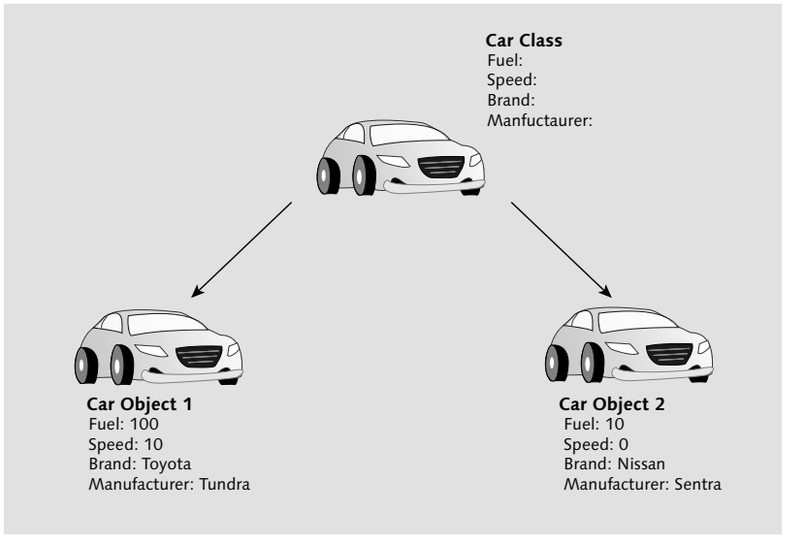


Figure 6.4 Relationship between Class and Object

Modularizing with Object-Oriented Programming

Just because you are using object-oriented programming doesn't mean you have to use it to build multiple objects. You could also have

one object that holds all of the logic for your program. Each method will represent a single function, as discussed in the section on the separation of concerns principle. Looking back at Figure 6.2, each different unit could be represented as a method in a flight finder class, as shown in Figure 6.5.

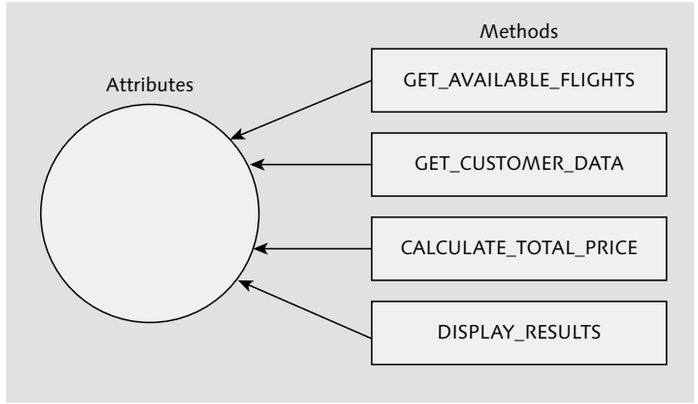


Figure 6.5 Flight Finder Class Concept

Passing data to methods

Each class method can be created with the ability to take and return data. For example, when creating the method for calculating the total price in Figure 6.5, you could pass a value containing the results from the get available flights method.

Structuring Classes

Now that you understand some of the concepts of object-oriented programming, you can begin to learn how to create classes and objects in ABAP. If the object-oriented concepts do not make sense yet, perhaps seeing the actual code in action will help. We'll first cover how to create a local class within a program and then how to create a global class that can be used across multiple programs.

Implementation vs. Definition

In ABAP, every class requires a definition and an implementation. The definition lists the different attributes and methods of a class,

whereas the implementation contains the actual code for all of the methods. The definition must come before the implementation and must also come before an object is created from the class. The class is defined by using the `CLASS` keyword, followed by the name of the class and then either `DEFINITION` or `IMPLEMENTATION` depending on what you are declaring. The example in Listing 6.2 contains the definition of a class with no attributes or methods. Prefix the class name with `lcl` to indicate that it's a local class, meaning that it's being created inside of an ABAP Program. Since we are demonstrating local classes, you can insert the code in this section into any ABAP program for testing. Since both the definition and implementation are contained within a `CLASS` and `ENDCLASS` keyword, they do not need to be next to each other when being defined.

```
CLASS lcl_car DEFINITION.
ENDCLASS.
CLASS lcl_car IMPLEMENTATION.
ENDCLASS.
```

Listing 6.2 Definition and Implementation of a Class

Creating Objects

Now that you've created a basic class, you can create objects of that class. Remember that a class is like a design, and you can build multiple objects using that design.

Object variables

There are two parts to creating an object. The first part is to define the object variable. This is just like creating variables, which we introduced in Chapter 2, except that you will use `TYPE REF TO` instead of just `TYPE` to indicate the class to be used when creating the object. The example in Listing 6.3 uses the prefix `o_` to indicate an object.

For objects, you then use the command `CREATE OBJECT` followed by the object variable to instantiate the object, as shown in Listing 6.3 Just like creating your own data types using the `TYPES` command, the class definition must come before creating an object using that class.

```
CLASS lcl_car DEFINITION.
ENDCLASS.

DATA: o_car TYPE REF TO lcl_car.
CREATE OBJECT o_car.
```



```
CLASS lcl_car IMPLEMENTATION.
ENDCLASS.
```

Listing 6.3 Creating an Object from a Class

Public and Private Sections

Before adding attributes and methods to the class definition, you will need to decide whether those attributes and methods should be public, private, or protected.

Public

Public attributes and methods can be used within the class or outside of the class, from the main program, or even from another class.

Private

Private attributes and methods can only be used from within the class itself, meaning that another class or the main program is unable to read the attributes or call the methods that are listed as private.

Protected

Protected attributes and methods can only be used from within the class itself, just like the private attributes and methods. The difference with protected attributes and methods is that they can be inherited from a subclass, unlike a private attribute or class. We will revisit protected attributes and methods when we cover inheritance later in the chapter.

The public and private sections are defined in the class implementation using the `PUBLIC SECTION` and `PRIVATE SECTION` keywords. Listing 6.4 adds those sections to the class definition.

```
CLASS lcl_car DEFINITION.
PUBLIC SECTION.
PRIVATE SECTION.
ENDCLASS.
CLASS lcl_car IMPLEMENTATION.
ENDCLASS.
```

Listing 6.4 Adding the Public and Private Sections to the Class Definition

Attributes

Next, you can add attributes to public or private sections, by creating variables, just like the ones discussed in Chapter 2. The variables must be defined after a section to determine whether they're public or private.

These attributes will be available globally to all of your methods; if they're public, they'll also be available globally outside of your

methods. Listing 6.5 adds public attributes for fuel, speed, brand, and manufacturer and a private attribute for the current gear.

Read Only attributes

Public attributes can also be given a `READ-ONLY` property, which will make them readable outside of the class and changeable only from within the class. In Listing 6.5, the attribute `d_manufacturer` is set to be read-only.

```
CLASS lcl_car DEFINITION.
PUBLIC SECTION.
     DATA: d_fuel TYPE i,
           d_speed TYPE i,
           d_brand TYPE string,
           d_manufacturer TYPE string READ-ONLY.
PRIVATE SECTION.
     DATA: d_gear TYPE i.
ENDCLASS.
CLASS lcl_car IMPLEMENTATION.
ENDCLASS.
```

Listing 6.5 Adding Public and Private Attributes to the Car Class

Class Methods

Definition

Methods are the place to store all of your code, in single units of work designed to complete one function. Each method should have a name describing the action that it will complete. First, you have to define the method using the keyword `METHODS` in the class definition, and then write the method's code in the class implementation in between the words `METHOD` and `ENDMETHOD`. Listing 6.6 expands on the car example to add the definition and an empty implementation for the accelerate, decelerate, and refuel methods. When adding methods, you will get a syntax error if the method is not defined in both the definition and implementation of the class.

```
CLASS lcl_car DEFINITION.
PUBLIC SECTION.
     DATA: d_fuel TYPE i,
           d_speed TYPE i,
           d_brand TYPE string,
           d_manufacturer TYPE string.
      METHODS: accelerate,
               decelerate,
               refuel.
PRIVATE SECTION.
     DATA: d_gear TYPE i.
```



```
ENDCLASS.
CLASS lcl_car IMPLEMENTATION.
     METHOD accelerate.
     ENDMETHOD.
     METHOD decelerate.
     ENDMETHOD.
     METHOD refuel.
     ENDMETHOD.
ENDCLASS.
```

Listing 6.6 Adding Public Classes to the Car Class

Implementation

Now, you can add code to the methods. The code will go in the class implementation section, and each method will share all of the attributes declared in the definition. Listing 6.7 adds some programming logic to each of the methods, and you can see that they are all able to access the class attributes.

Local variables

You can also declare variables within methods, such as `ld_max` in the `REFUEL` method shown in Figure 6.11. These variables are considered local variables since they will not be visible or usable outside of the methods in which they're declared. For that reason, they are prefixed with a `ld_` meaning local data variable, instead of `d_`, meaning global data variable.

```
CLASS lcl_car DEFINITION.
PUBLIC SECTION.
     DATA: d_fuel TYPE i,
           d_speed TYPE i,
           d_brand TYPE string,
           d_manufacturer TYPE string.
      METHODS: accelerate,
               decelerate,
               refuel.
PRIVATE SECTION.
     DATA: d_gear TYPE i.
ENDCLASS.
CLASS lcl_car IMPLEMENTATION.
     METHOD accelerate.
          d_speed = d_speed + 5.
          d_fuel = d_fuel - 5.
     ENDMETHOD.
     METHOD decelerate.
          d_speed = d_speed - 5.
          d_fuel = d_fuel - 2.
     ENDMETHOD.
     METHOD refuel.
```

```
          DATA: ld_max TYPE I VALUE 100.
          d_fuel = ld_max.
     ENDMETHOD.
ENDCLASS.
```

Listing 6.7 Adding Logic to Methods in the Car Class

Calling methods

Now that you've defined some methods which contain some code, you can define an object, create it, and call the methods contained in the object. To call an object's method, you enter the object name followed by an arrow (`->`), then the method name, and then the open and close parentheses (`()`), with any parameters in between the parentheses. Table 6.1 shows how a method is called via an example using the car object created earlier in this section.

`Object->method_name(parameters)`	`o_car->accelerate()`

Table 6.1 How to Call a Method

The code for calling methods has to be included after the object has been defined and created, but it can come before the definition of the object, as shown in Listing 6.8.

```
CLASS lcl_car DEFINITION.
PUBLIC SECTION.
     DATA: d_fuel TYPE i,
           d_speed TYPE i,
           d_brand TYPE string,
           d_manufacturer TYPE string.
      METHODS: accelerate,
               decelerate,
               refuel.
PRIVATE SECTION.
     DATA: d_gear TYPE i.
ENDCLASS.
DATA: o_car TYPE REF TO lcl_car.
CREATE OBJECT o_car.
o_car->accelerate( ).
CLASS lcl_car IMPLEMENTATION.
     METHOD accelerate.
          d_speed = d_speed + 5.
          d_fuel = d_fuel - 5.
     ENDMETHOD.
     METHOD decelerate.
          d_speed = d_speed - 5.
          d_fuel = d_fuel - 2.
     ENDMETHOD.
     METHOD refuel.
          DATA: d_max TYPE i VALUE 100.
          d_fuel = 100.
     ENDMETHOD.
ENDCLASS.
```

Listing 6.8 How to Call a Method in an Object

Methods in the execution stack

Now you can set a breakpoint within the method and execute the program and you will see the execution stack will show where the method was called from. If you are using Eclipse, the stack should look like what we see in Figure 6.6. The `accelerate` method is listed and before that is `START-OF-SELECTION`, which is the ABAP event that starts executing our code.

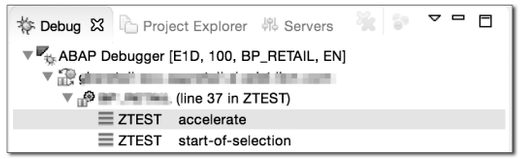


Figure 6.6 Execution Stack in Eclipse

Now, we can select the `start-of-selection` item in the stack to see the line of code where the `accelerate` method was called as shown in Figure 6.7.

Figure 6.7 Eclipse Showing Where the Start-Of-Selection Stack

We can do the same thing from the SAP GUI debugger, when your breakpoint inside of the `accelerate` method is hit, you will notice the ABAP AND SCREEN STACK section will appear as shown in Figure 6.8.

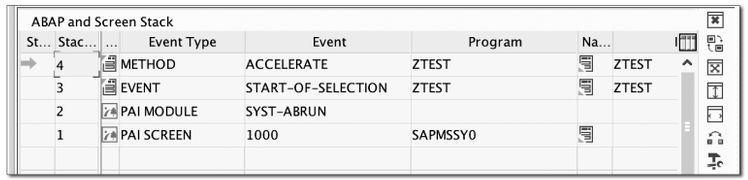


Figure 6.8 ABAP and Screen Stack in SAP GUI

We can then click the START-OF-SELECTION item and the debugger will bring up the section of code that called our accelerate method as shown in Figure 6.9.

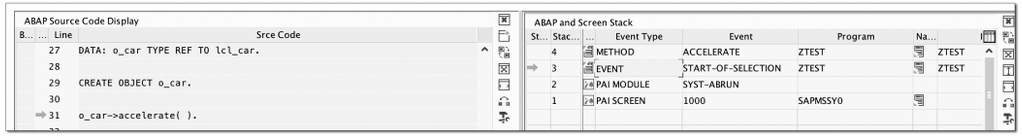


Figure 6.9 SAP GUI Debugger Execution Stack

Importing, Returning, Exporting, and Changing

When using your car class, users do not want to accelerate at a rate of 5; they want to specify the amount of acceleration to occur, which makes sense. There are a few ways to pass data to and from methods, as described in Table 6.2.

Importing	A copy of one or more variables is passed to the method.
Returning	The actual variable is returned by the method. Returning can only be used to return one variable.
Exporting	A copy of one or more variables are returned from the method.
Changing	The actual variable is passed to the method, and any changes to that variable will change the original. (Also known as passing by reference.)

Table 6.2 Ways to Pass Data to and from a Method

Importing

You can change the `accelerate` method to import a variable to indicate the amount of speed that you want to increase by. This is handled in Listing 6.9 by adding the `IMPORTING` command followed by a variable definition for the variable that will be copied in to the method and used to set the rate of acceleration for the car object. After adding the `IMPORTING` variable in the definition, that variable can now be accessed in the method implementation. The prefix `ip` here indicates an `IMPORTING` parameter.

Now that you've defined the `IMPORTING` parameter, you also can pass the value for that parameter within the parentheses when calling the method, as shown in Listing 6.9.


```
CLASS lcl_car DEFINITION.
PUBLIC SECTION.
     DATA: d_fuel TYPE i,
           d_speed TYPE i,
           d_brand TYPE string,
           d_manufacturer TYPE string.
      METHODS: accelerate IMPORTING ip_accel_rate TYPE i,
               decelerate,
               refuel.
PRIVATE SECTION.
     DATA: d_gear TYPE i.
ENDCLASS.
DATA: o_car TYPE REF TO lcl_car.
CREATE OBJECT o_car.
o_car->accelerate( 5 ).
CLASS lcl_car IMPLEMENTATION.
     METHOD accelerate.
          d_speed = d_speed + ip_accel_rate.
          d_fuel = d_fuel - 5.
     ENDMETHOD.
     METHOD decelerate.
          d_speed = d_speed - 5.
          d_fuel = d_fuel - 2.
     ENDMETHOD.
     METHOD refuel.
          d_fuel = 100.
     ENDMETHOD.
ENDCLASS.
```

Listing 6.9 Adding the Ability to Import Variables in Methods

Returning

Next, you can change the method to check if the fuel is at zero; if it is, then the car will not accelerate. When you call the method, you want to know if it worked or not, so return a Boolean parameter that will be true if the method worked and false if it did not. The Boolean parameter is defined in the class definition using the `RETURNING` keyword followed by `VALUE` and the variable name within parentheses, as shown in Listing 6.10, with the prefix `rp` indicating a returning parameter.

Because the returning parameter is a Boolean, also create the Boolean variable `d_is_success` in the main program and set it to the result of the method call in Listing 6.10, meaning that `d_is_success` will be set to the value of `rp_is_success` after calling the `accelerate` method.

```
CLASS lcl_car DEFINITION.
PUBLIC SECTION.
     DATA: d_fuel TYPE i,
           d_speed TYPE i,
           d_brand TYPE string,
           d_manufacturer TYPE string.
      METHODS: accelerate IMPORTING ip_accel_rate TYPE i
               RETURNING VALUE(rp_is_success)
                 TYPE bool,
               decelerate,
               refuel.
PRIVATE SECTION.
     DATA: d_gear TYPE i.
ENDCLASS.
DATA: o_car TYPE REF TO lcl_car,
      d_is_success TYPE bool.
CREATE OBJECT o_car.
d_is_succcess = o_car->accelerate(5).
ClASS lcl_car IMPLEMENTATION.
     METHOD accelerate.
          IF d_fuel - 5 > 0.
              d_speed = d_speed + ip_accel_rate.
              d_fuel = d_fuel - 5.
              rp_is_success = abap_true.
          ELSE.
               rp_is_success = abap_false.
          ENDIF.
     ENDMETHOD.
     METHOD decelerate.
          d_speed = d_speed - 5.
          d_fuel = d_fuel - 2.
     ENDMETHOD.
     METHOD refuel.
          d_fuel = 100.
     ENDMETHOD.
ENDCLASS.
```

Listing 6.10 Class Including Returning Parameter

Method chaining

The returned variable can also be used in line with other ABAP keywords using *method chaining*, which was added to the ABAP language in ABAP 7.02. For example, you can call the `accelerate` method from Listing 6.10 from within an `IF` statement and use the returning parameter as part of the `IF` statement, as shown in Listing 6.11.

```
…
DATA: o_car TYPE REF TO lcl_car,
      d_is_success TYPE bool.
```



```
CREATE OBJECT o_car.
IF o_car->accelerate(5) = abap_true.
     WRITE: 'It worked!'.
ENDIF.
…
```

Listing 6.11 Using the Returning Parameter as Part of an IF Statement

Import multiple parameters

If you want to import multiple parameters, you can do so by including the additional parameters after the `IMPORTING` command in the class definition. When you list multiple variables, you will also need to specify which variable you are passing within the parentheses, as shown in Listing 6.12.

```
CLASS lcl_car DEFINITION.
PUBLIC SECTION.
     DATA: d_fuel TYPE i,
           d_speed TYPE I,
           d_brand TYPE string,
           d_manufacturer TYPE string.
      METHODS: accelerate IMPORTING ip_accel_rate TYPE i
                                    ip_other_param TYPE i
               RETURNING VALUE(rp_is_success)
                 TYPE bool,
               decelerate,
               refuel.
PRIVATE SECTION.
     DATA: d_gear TYPE i.
ENDCLASS.
DATA: o_car TYPE REF TO lcl_car,
      d_is_success TYPE bool.
CREATE OBJECT o_car.
d_is_succcess = o_car->accelerate( ip_accel_rate = 5
  ip_other_param = 1 ).
ClASS lcl_car IMPLEMENTATION.
..
ENDCLASS.
```

Listing 6.12 Adding Multiple IMPORTING Parameters

Optional parameters

You can also mark parameters as optional by adding `OPTIONAL` after the parameter's definition, which means that the parameter will have an initial value when the method runs if a value is not entered for that parameter. If there are multiple importing parameters but only one is not marked as optional, you can pass only the required parameter without identifying the parameter names, as shown in Listing 6.13.

```
CLASS lcl_car DEFINITION.
PUBLIC SECTION.
     DATA: d_fuel TYPE i,
           d_speed TYPE i,
           d_brand TYPE string,
           d_manufacturer TYPE string.
      METHODS: accelerate IMPORTING ip_accel_rate TYPE i
                                    ip_other_param TYPE i OPTIONAL
                  RETURNING VALUE(rp_is_success)
                    TYPE bool,
                  decelerate,
                  refuel.
PRIVATE SECTION.
     DATA: d_gear TYPE i.
ENDCLASS.
DATA: o_car TYPE REF TO lcl_car,
      d_is_success TYPE bool.
CREATE OBJECT o_car.
d_is_succcess = o_car->accelerate( 5 ).
CLASS lcl_car IMPLEMENTATION.
..
ENDCLASS.
```

Listing 6.13 Optional Importing Parameters

Returning multiple parameters

The RETURNING parameter only allows you to return one parameter, but if you need to return multiple parameters, you can use EXPORTING parameters. EXPORTING parameters are defined just like IMPORTING parameters, but using both requires identifying which parameters are IMPORTING and which are EXPORTING when calling the corresponding method, as shown in Listing 6.14.

Exporting

The parameters that are EXPORTING from the method will be IMPORTING into the main program; note how the naming changes in Listing 6.14. You cannot use both EXPORTING and RETURNING in the same method definition. EXPORTING parameters are always optional.

```
CLASS lcl_car DEFINITION.
PUBLIC SECTION.
     DATA: d_fuel TYPE i,
           d_speed TYPE I,
           d_brand TYPE string,
           d_manufacturer TYPE string.
      METHODS: accelerate IMPORTING ip_accel_rate TYPE i
                          EXPORTING ep_is_success TYPE bool,
                  decelerate,
                  refuel.
```



```
PRIVATE SECTION.
     DATA: d_gear TYPE i.
ENDCLASS.
DATA: o_car TYPE REF TO lcl_car,
      d_is_success TYPE bool.
CREATE OBJECT o_car.
o_car->accelerate( EXPORTING ip_accel_rate = 5
                   IMPORTING ep_is_success = d_is_success ).
CLASS lcl_car IMPLEMENTATION.
..
ENDCLASS.
```

Listing 6.14 EXPORTING Parameters

Changing

The last option for passing variables to or from methods is CHANGING. CHANGING parameters are passed into a method like IMPORTING parameters but can also be changed and returned like EXPORTING parameters. Unlike with an IMPORTING parameter, when calling a method and using a CHANGING parameter, you will always have to specify that it is a CHANGING parameter. Just as with EXPORTING parameters, you can't have both a CHANGING and a RETURNING parameter. The car object example has been updated to use a CHANGING parameter in Listing 6.15.

```
CLASS lcl_car DEFINITION.
PUBLIC SECTION.
     DATA: d_fuel TYPE i,
           d_speed TYPE I,
           d_brand TYPE string,
           d_manufacturer TYPE string.
      METHODS: accelerate IMPORTING ip_accel_rate TYPE i
                          CHANGING cp_is_success TYPE bool,
                  decelerate,
                  refuel.
PRIVATE SECTION.
     DATA: d_gear TYPE i.
ENDCLASS.
DATA: o_car TYPE REF TO lcl_car,
      d_is_success TYPE bool.
CREATE OBJECT o_car.
o_car->accelerate( EXPORTING ip_accel_rate = 5
                   CHANGING  cp_is_success = d_is_success ).
CLASS lcl_car IMPLEMENTATION.
..
ENDCLASS.
```

Listing 6.15 CHANGING Parameters

Returning vs. Exporting/Changing

When possible, use `RETURNING` parameters to help write more concise, easier-to-read code. `RETURNING` parameters will also make your ABAP code look more similar to other object-oriented languages, allowing someone who learned other languages first to more easily understand your program. The real power of `RETURNING` parameters is the ability to use method chaining with other ABAP keywords, as shown in Listing 6.11.

Constructors

Constructors are special methods used to set the state of an object before you start calling its methods. For example, you can set the manufacturer and model of the car object when you first create it using a constructor.

Creating a constructor

A constructor is created by creating a method called `constructor`. This method will be called by the `CREATE OBJECT` keyword, and any required parameters must be passed when using `CREATE OBJECT`. The example in Listing 6.16 specifies the model and manufacturer in the constructor, and those parameters are used to update the class attributes.

```
CLASS lcl_car DEFINITION.
PUBLIC SECTION.
     DATA: d_fuel TYPE i,
           d_speed TYPE i,
           d_model TYPE string,
           d_manufacturer TYPE string.
      METHODS: accelerate IMPORTING ip_accel_rate TYPE i
           RETURNING VALLUE(rp_is_success)
             TYPE bool,
           decelerate,
           refuel,
           constructor
           IMPORTING ip_manufacturer TYPE string
                     ip_model        TYPE string.
PRIVATE SECTION.
     DATA: d_gear TYPE i.
ENDCLASS.
DATA: o_car TYPE REF TO lcl_car,
      d_is_success TYPE bool.
```



```
CREATE OBJECT o_car EXPORTING ip_manufacturer = 'Toyota'
                              ip_model        = 'Tundra'.
d_is_success = o_car->accelerate( 5 ).
CLASS lcl_car IMPLEMENTATION.
METHOD constructor.
     d_manufacturer = ip_manufacturer.
     d_model        = ip_model.
ENDMETHOD.
..
ENDCLASS.
```

Listing 6.16 Car Object with a Constructor

The constructor is typically used to set variables that determine the state of the object, but it can be used to initialize the object in any way. For example, the constructor can read data from a database table to fill some private or public variables. Remember that the constructor will only be run once when creating the object.

Recursion

When using separation of concerns, you can use *recursion* to reuse a unit of work thanks to method chaining, which we introduced in Listing 6.11. Recursion works by calling a method from inside the same method, which can make your code much more compact and understandable than following a procedural approach, which may just result in one long program with the same code repeated over and over. Because you call a method from inside the same method using the results of the earlier call, recursion only works when you use a `RETURNING` parameter in your method.

Fibonacci sequence

To demonstrate recursion, let's write a method to calculate a given number of the Fibonacci sequence. The Fibonacci sequence is an infinite sequence starting with 0 and 1 and then adding the current number to the last number to get the next number in the sequence. For example: 0, 1, 1, 2, 3, 5, 8, and 13. If you are familiar with agile scrum planning, you may have used Fibonacci numbers to estimate work.

In Listing 6.17, to calculate the Fibonacci number you recursively call the `calculate` method, meaning that we call it from within the `calculate` method and set the returning value to its result. During this

call, call with a value of one less than the current importing parameter and add the result of that to the result of calling the method with a value of two less than the current importing parameter. You don't want the recursive call to run forever, of course, so if the method is called with a value of one or less, it will return the value of the parameter that was called. Try running the code in Listing 6.17 in debug mode and step through the program to see the recursion in action.

```
CLASS lcl_fibonacci DEFINITION.
PUBLIC SECTION.
     METHODS: calculate IMPORTING ip_place TYPE i
                        RETURNING VALUE(rp_value) TYPE i.
ENDCLASS.
DATA: o_fib TYPE REF TO lcl_fibonacci,
      d_result TYPE i.
CREATE OBJECT o_fib.
d_result = o_fib->calculate( 4 ).
CLASS lcl_fibonacci IMPLEMENTATION.
METHOD calculate.
     IF ip_place <= 1.
          rp_value = ip_place.
     ELSE.
          rp_value = calculate( ip_place -
 1 ) + calculate( ip_place - 2 ).
     ENDIF.
ENDMETHOD.
ENDCLASS.
```

Listing 6.17 Calculating a Fibonacci Number with Recursion

Figure 6.10 provides a conceptual image of what is happening during the recursive call in Listing 6.17 when trying to find the fourth number of the Fibonacci sequence. Because the code only returns the importing parameter value when it is 1 or less, the numbers added up are always 1 or 0.

A need for a recursive method will not exist in every ABAP program that you write, but when it can be used, it's very useful and can make your program much more understandable.

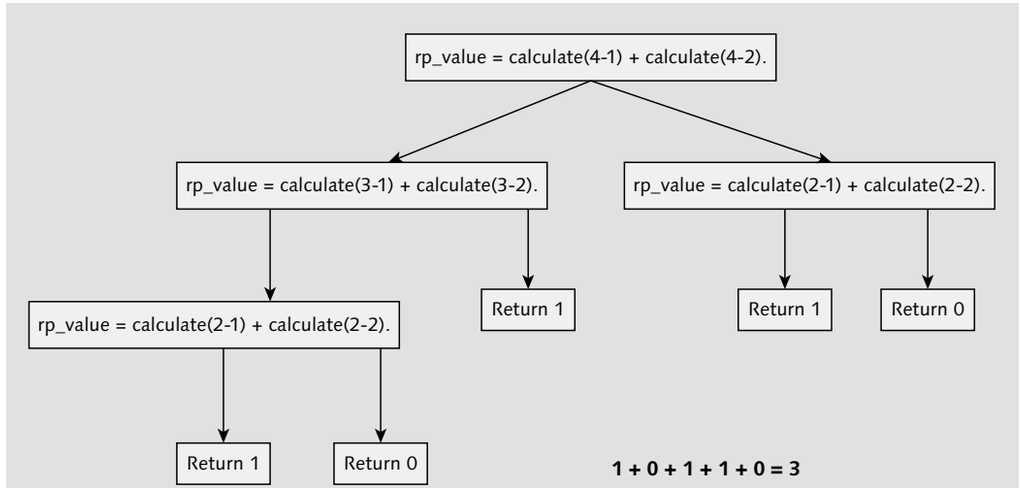


Figure 6.10 Fibonacci Recursive Call

Inheritance

When using object oriented programming, we have the ability to take on attributes and methods from another class through the use of *inheritance*. The class that we are inheriting from is called a *superclass* and the class that is inheriting is called a *subclass*.

Inheritance, like recursion, is a powerful benefit of object-oriented programming that we do not necessarily use with every program that we write.

INHERITING FROM

In Listing 6.18, we added a new class, called `lcl_truck` which inherits from the car class by using the `INHERITING FROM` keyword as part of the class definition. Since `lcl_truck` inheriting from `lcl_car`, we can call all of the same methods that are in `lcl_car` and it will execute the code in the `lcl_car` implementation. This class can be defined below the code that we have already covered.

```
…
CLASS lcl_truck DEFINITION
INHERITING FROM lcl_car.
ENDCLASS.

DATA: o_truck TYPE REF TO lcl_truck.
```

```
CREATE OBJECT o_truck.
o_truck->accelerate(5).

CLASS lcl_truck IMPLEMENTATION.
ENDCLASS.
```

Listing 6.18 Demonstrating Class Inheritance

Redefining methods

We also have the option of redefining a method so that we use the code inside the `lcl_truck` implementation instead of the `lcl_car` implementation. To do that, we need to declare the method in the subclass without any parameters and include the `REDEFINITION` keyword as shown in Listing 6.19. Notice that we still have access to the same public variables and the returning parameter that we were able to use in the superclass method.

```
…
CLASS lcl_truck DEFINITION
INHERITING FROM lcl_car.
 PUBLIC SECTION.
         METHODS: accelerate REDEFINITION.
ENDCLASS.

DATA: o_truck TYPE REF TO lcl_truck.
CREATE OBJECT o_truck.

o_truck->accelerate(5).

CLASS lcl_truck IMPLEMENTATION.
    METHOD accelerate.
        d_speed = 1.
        rp_is_success = abap_true.
    ENDMETHOD.
ENDCLASS.
```

Listing 6.19 Demonstrating Redefining a Method in a Subclass

Protected section

Unlike methods in the superclass, methods in the subclass are unable to access the private methods or attributes of the superclass. If you want to create attributes or methods that are private but can also be accessed from inside of a subclass, then you must define them in the protected section. In Listing 6.20 we demonstrate this by adding a protected variable to the `lcl_car` definition which we can then call from within the `lcl_truck` class.


```
CLASS lcl_car DEFINITION.
PUBLIC SECTION.
     DATA: d_fuel TYPE i,
           d_speed TYPE i,
           d_model TYPE string,
           d_manufacturer TYPE string.
      METHODS: …
PRIVATE SECTION.
     DATA: d_gear TYPE i.
PROTECTED SECTION.
     DATA: d_protected TYPE i.
ENDCLASS.
…
CLASS lcl_truck DEFINITION
INHERITING FROM lcl_car.
 PUBLIC SECTION.
         METHODS: accelerate REDEFINITION.
ENDCLASS.
…
CLASS lcl_truck IMPLEMENTATION.
    METHOD accelerate.
        d_speed = 1.
        rp_is_success = abap_true.
        d_protected = 1.
    ENDMETHOD.
ENDCLASS.
```

Listing 6.20 Class Demonstrating a Protected Attribute

Global Classes

So far, you've learned about local classes, which are created and run inside of a local program. However, there are many use cases in which you'll want to use classes across multiple programs. A common use of global classes is to create interfaces with custom tables that you have written. This allows for a couple of things: First, you can handle locking of tables within the methods of your class, instead of having to lock and unlock tables in the programs that access the table. Second, you can write methods that allow you to access the data in your custom table(s) without having to write your own `SELECT` statements. How you use global classes really depends on the design of your tables and applications and the problem that you're trying to solve.

Multiple views

The code for creating and changing the classes is mostly the same as what you saw for local classes. When using SAP GUI to create the class, there are two views: a source code view and a forms view. The resulting code will be the same, but the forms view allows the ABAP system to write some of the code itself.

Source control

One thing that is different about using global classes is the way that they're broken up into different pieces. Each piece has its own source control history and must be activated on its own. The breakup of a class into different pieces makes it possible for multiple developers to work on the same class simultaneously.

The different pieces of a global class are the public, private, and protected sections and each method implementation. This allows you to treat each method as an individual program in terms of source control and activation. The public, private, and protected sections are a bit of a special case, however. These sections are automatically generated when using the forms view in SAP GUI, so any comments entered in these areas will be overwritten. You can still create and edit these sections with your own code, which will be kept, but any comments will be lost.

How to Create Global Classes in Eclipse

Using Eclipse to create classes is the preferred method and we'll look at that first, but we will also cover how to create them using Transaction SE80 later in the section.

To begin, select File • New • ABAP Class; you'll see the New ABAP Class wizard appear. Select the package $TMP to save the new class as a local object, enter "ZCL_GLOBAL_CLASS" as the class name, and enter "Global Class" as the description, as shown in Figure 6.11. Just as with ABAP programs, your class needs to be prefixed with a Z. You can use the prefix ZCL to indicate that the class you're creating is a custom global class.

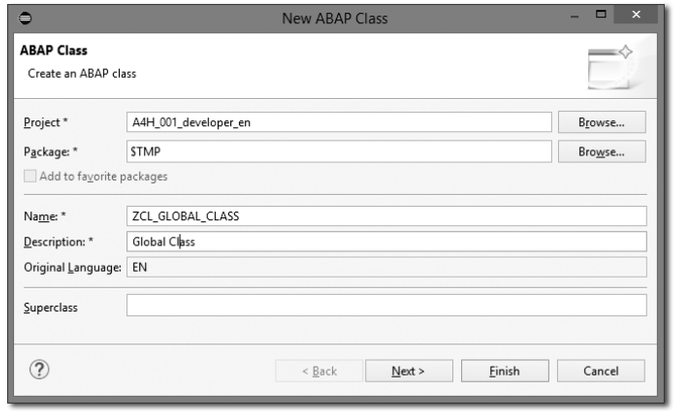


Figure 6.11 Creating a New Global Class in Eclipse

Global class keywords

You'll see your new class with the basic structure for a class laid out for you, as shown in Figure 6.12. This should look familiar; it's the same structure that you saw with the local classes. The only change is the addition of the `PUBLIC` keyword, indicating a global class. Additionally, `FINAL` keyword is optional and indicates that the class cannot be inherited from and the `CREATE PUBLIC` keyword, allows an object to be created from the class anywhere where the class is visible.

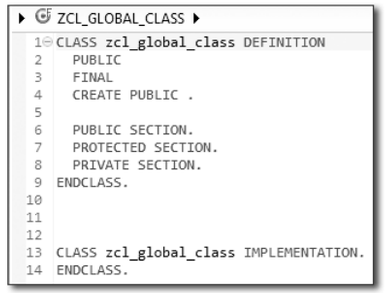


Figure 6.12 New Class Created in Eclipse

How to Create Global Classes in Transaction SE80

If you're using Transaction SE80 as your ABAP IDE, select CLASS/INTERFACE from the dropdown in the center of left side of the screen, type "ZCL_GLOBAL_CLASS" in the textbox below the dropdown in the center left of the screen, and press Enter.

You will be prompted with a popup asking if you want to create ZCL_GLOBAL_CLASS because it doesn't exist as shown in Figure 6.13. Click the YES button.

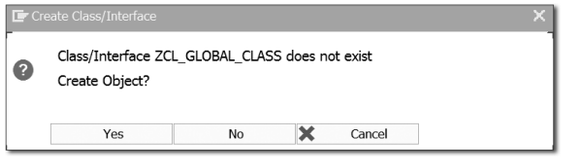


Figure 6.13 Create Class/Interface Popup

In the next popup, enter "Global Class" in the DESCRIPTION field and leave the rest of the options set to their defaults, as shown in Figure 6.14. The FINAL checkbox indicates that other classes cannot inherit from this class. Select the SAVE button to continue.

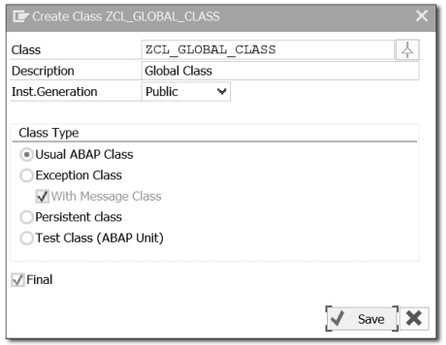


Figure 6.14 Create Class Popup

In the CREATE OBJECT DIRECTORY ENTRY popup, click the LOCAL OBJECT button or enter "$TMP" for the package name, and click the SAVE button. For production objects, you should use a package created for your project.

Forms view

After creating the class, the editor will show the class in the default forms view. The forms view can be used to change the structure of the class and add attributes and methods, and it will generate the class definition code automatically. You can also click the SOURCE CODE-BASED button (Figure 6.15) to view the entire class as code.

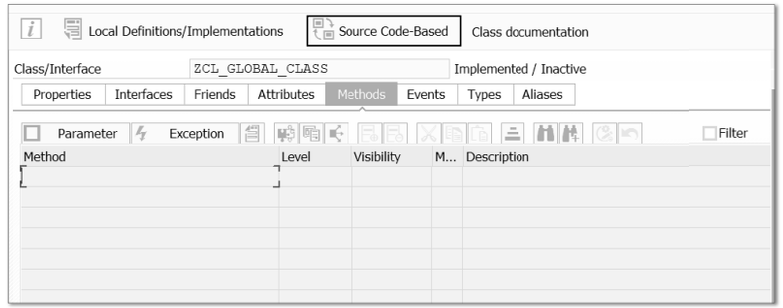


Figure 6.15 Forms View of a Class, with Source Code-Based Button Highlighted

Source code view

The source code view shown in Figure 6.16 should look similar to what you saw when working with local classes. Whichever method you choose to use should be based on personal preference, but the source code view is recommended if you are going to use Transaction SE80 as your primary IDE.

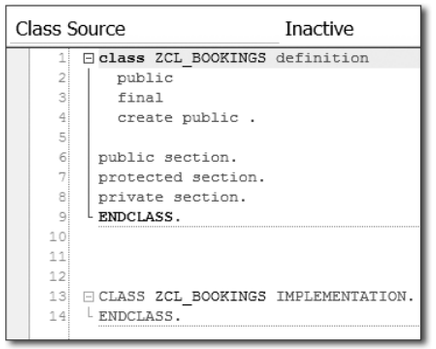


Figure 6.16 New Class in the Source Code View

Using the Form-Based View in Transaction SE80

You can now return to the form-based view by clicking the FORM-BASED button, if you are still looking at the source code view. The form-based view will generate the code for the class definition and add the method definitions to the class implementation. Everything in the form-based view can be done manually in the source code view; which you use is a matter of personal preference.

There are many things that you can do with classes that I haven't covered yet and things that are out of scope for this book, so don't feel overwhelmed by all of the tabs and options in the form-based view. In this section, you will learn how to use the form-based view to add methods and attributes to a class.

Creating a method

First, create a new method. To do so, select the METHODS tab and enter "METHOD1" in the first row of the METHOD column; this will name your new method `METHOD1`. Next, under LEVEL, select INSTANCE METHOD. When you worked with local classes earlier in the chapter, those were instance methods; static methods are out of scope for this book and should be avoided if possible.

Next, select PUBLIC under VISIBILITY; this will set the method as public by defining it in the public section of the class definition. You can also add a description in the last column, which will only be visible from the form-based view. The name of your method should typically be descriptive enough on its own. The end result should look as shown in Figure 6.17.

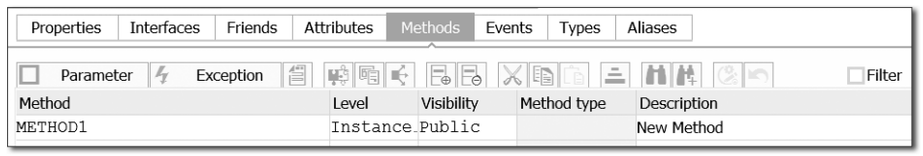


Figure 6.17 Adding a Method Using the Form-Based View

Parameters

Next, select the method and click the PARAMETER button. From here, you can add parameters for your method. The parameter name goes in the first column; enter "ip_parameter". The next column, TYPE, indicates whether the parameter is importing, exporting, returning, or changing. Select IMPORTING for this parameter.

Next, there are two checkboxes, one for PASS VALUE and one for OPTIONAL. You're required to pass a value for returning parameters, but can pass a value for any importing parameters as well. Passing a value for a parameter means that any changes to the parameter in the method will change the parameter that was passed into the method instead of changing a copy of that parameter. The OPTIONAL checkbox will make that parameter optional.

The next column, TYPING METHOD, indicates whether the data type should be created using the keyword `TYPE`, `TYPE REF TO`, or `LIKE`. Select the typing method TYPE. Next, the ASSOCIATED TYPE column indicates the data type to be used for the parameter; enter "I" to indicate an integer data type. You can also add a default value and description; as with the method description, this description is only visible to other developers using the form-based view.

Next, add a second parameter named "rp_value". This parameter should be of type returning with a typing method of `TYPE` and an associated type of `i` for integer. Both parameters are shown in Figure 6.18.

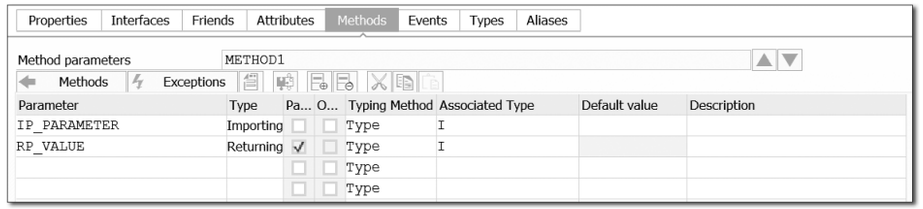


Figure 6.18 Adding Parameters Using the Form-Based View

Method implementation

Now, click the METHODS button to return to the methods list, and double-click your method to enter the method implementation, where you can write your code. You will notice that the editor restricts you to only the method you selected. From here, you can also click the SIGNATURE button to toggle the signature display that will show the parameters you defined for the method as shown in Figure 6.19.

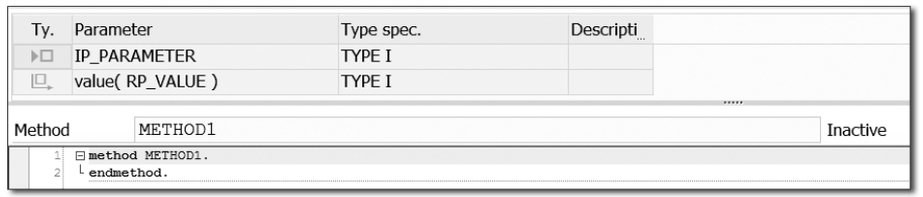


Figure 6.19 Editng a Method Using the Form-Based View

Attributes

Next, click BACK to return to the form-based view and click on the ATTRIBUTES tab to add some attributes to your new class. Enter "D_I"

in the first row of the ATTRIBUTE column to indicate that the attribute name will be `D_I`. Next, select INSTANCE ATTRIBUTE under the LEVEL column. Static attributes are out of scope for this book and should be avoided if possible.

Next, set VISIBILITY to PUBLIC to indicate that this attribute will be defined in the public section. The READ-ONLY checkbox will make the attribute read-only from outside of the class, but the attribute can still be changed within your methods. Next, set TYPING to TYPE and ASSOCIATED TYPE to I for integer.

The button to the right of the ASSOCIATED TYPE column is used to create complex types (Figure 6.20). Clicking that button will take you to the public, private, or protected section so that you can add your own custom attributes that aren't defined in the ABAP Data Dictionary.

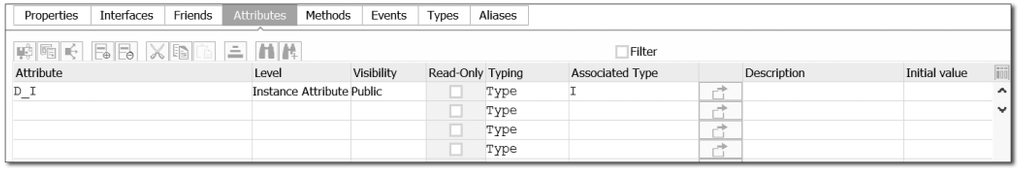


Figure 6.20 Adding Attributes Using the Form-Based View

Now that you've added a method and some variables using the form-based view, save your changes and click the SOURCE CODE-BASED VIEW button to see the automatically generated code (Figure 6.21). You will notice that the generated code still closely resembles what you saw with the local classes. The only differences are some added comments before the method and an added exclamation point (`!`) before the method parameters. The exclamation point is an escape symbol that allows you to use an ABAP keyword such as `RETURNING` as the name of a variable.

Any additional class examples in this book will only include the actual code, not the form-based view configuration. You should be familiar enough with classes by this point to use either the form-based view or the source code view interchangeably. We recommend using Eclipse or the source code view exclusively.

Class Source Inactive

```
1  class ZCL_GLOBAL_CLASS definition
2    public
3    final
4    create public .
5
6  public section.
7
8    data D_I type I .
9
10   methods METHOD1
11     importing
12       !IP_PARAMETER type I
13     returning
14       value(RP_VALUE) type I .
15 protected section.
16 private section.
17 ENDCLASS.
18
19
20
21 CLASS ZCL_GLOBAL_CLASS IMPLEMENTATION.
22
23
24 * <SIGNATURE>---------------------------------------------------------------------------------------+
25 * | Instance Public Method ZCL_GLOBAL_CLASS->METHOD1
26 * +--------------------------------------------------------------------------------------------------+
27 * | [--->] IP_PARAMETER                   TYPE        I
28 * | [<-()] RP_VALUE                       TYPE        I
29 * +--------------------------------------------------------------------------------------</SIGNATURE>
30   method METHOD1.
31   endmethod.
32 ENDCLASS.
```

Figure 6.21 Viewing the Code Generated by the Form-Based View

Obsolete Modularization

Now that you are familiar with modularizing your code using the modern object oriented approach, we will also cover some of the obsolete modularization techniques. You will need to be familiar with these techniques when working with old code or may need to use them for technical reasons.

Function Modules

Using function modules cannot be avoided in many ABAP programs, because some of SAP's standard functionality requires it in areas such as database table locks; however, you should never manually create new function modules. Any situation that calls for a function module to be created can use a global class instead. In fact, function modules were actually SAP's first attempt at making ABAP object oriented.

With that said, it's good to understand how function modules work, because they're prevalent in many customer systems and standard

SAP code and may be required for technical reasons, such as creating a remote function call (RFC).

Function groups

Similar to how methods are contained inside of a class, function modules are contained inside of a function group. The function group is created with two include files, one for global data (attributes in classes) and one that contains all of the function modules (methods in classes) in the group.

Any global data or function group attributes will hold the same value until the end of the program, but variables defined in the function module itself will only hold the same value until the end of the function call.

In your code, you call the function module itself instead of the group, and there's only one instance of the function group. This is unlike class objects, which can have multiple instances, as was illustrated by the car example in Figure 6.4.

Creating Function Groups and Modules in Eclipse

In Eclipse, you can create a new function group by selecting File • New • Other...; then expand the ABAP folder and select the option for ABAP Function Group in the popup as highlighted in Figure 6.22, and click Next.

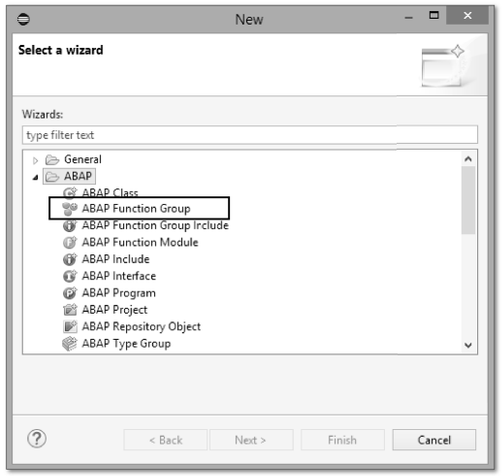


Figure 6.22 Creating a New ABAP Function Group

ABAP function group wizard

In the New ABAP Function Group wizard, enter the package "$TMP" to save it as a local object and enter "ZFG_FUNCTION_GROUP" for the function group name. The function group name will have to start with a Z as noted previously for programs and classes to indicate a custom function group. You can also prefix the name with "ZFG" to indicate that it is a function group. Remember that the function groups are like classes, so the name should indicate the type of object that the function group will be working with. Finally, enter a description, which may be used by other developers trying to find your function group. An example of the correct values is shown in Figure 6.23.

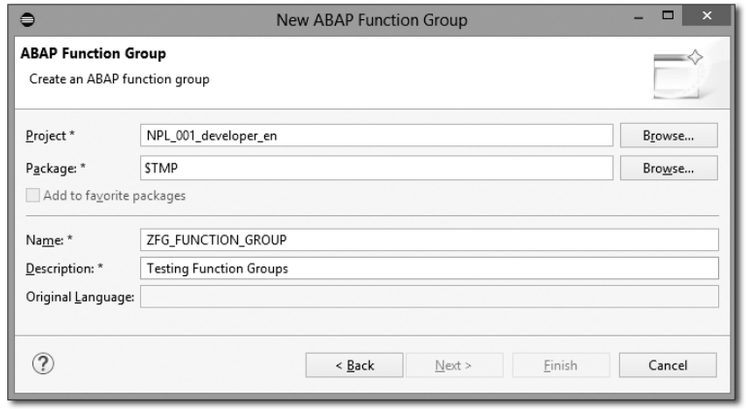


Figure 6.23 New ABAP Function Group Popup

INCLUDE

The function group code displayed in Figure 6.24 shows the two include files that make up the function group. The `INCLUDE` file ending in "`TOP`" will contain any global data variables that will be accessible by any of the function modules in the function group just like attributes of a class are available for all of the class methods. The `INCLUDE` ending in `UXX` will contain `INCLUDE`s for all of the function modules created, which are just like class methods.

Function module

To create the function module, select File • New • Other..., expand the ABAP folder, and select ABAP Function Module. You can also right-click the function group you just created from the Eclipse project explorer and select New • ABAP Function Module. This action will open the New ABAP Function Module popup (Figure 6.25).

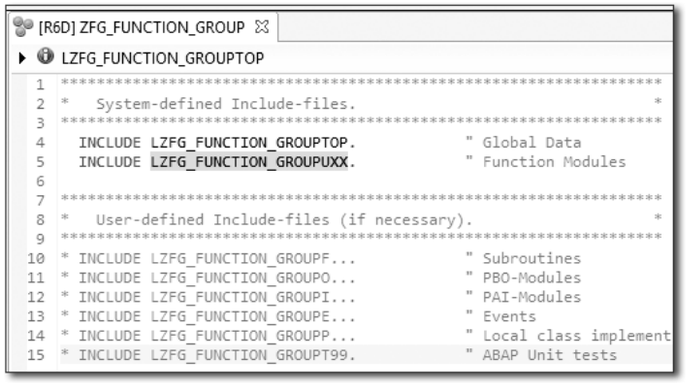


Figure 6.24 Code of a New Function Group

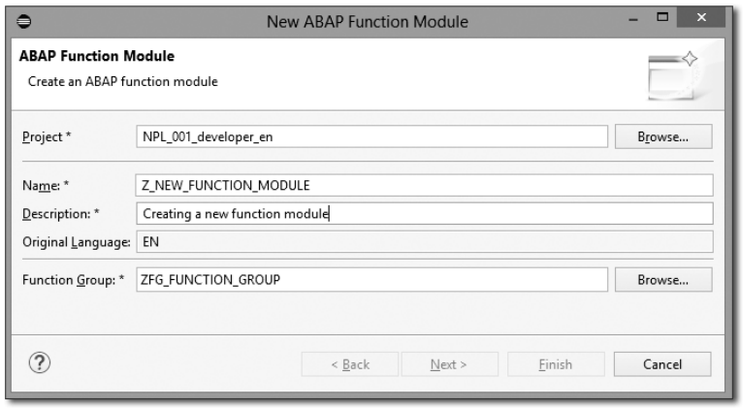


Figure 6.25 New ABAP Function Module Popup

From the popup, enter "Z_NEW_FUNCTION_MODULE" for the function module name. Because the code does not call the function group but instead calls the function module directly, the function module name is global, and each name can only be used once. A good practice is to prefix the function module name with something indicating the function group name after the required "Z_" prefix.

Next, enter a description that describes what the function module does to benefit other developers trying to find your function module. Finally, make sure that you enter the correct function group in which

the new function module should be contained in the FUNCTION GROUP textbox, and click NEXT and then the FINISH button.

The new function module should open, and you can enter any code in between the `FUNCTION` and `ENDFUNCTION` keywords.

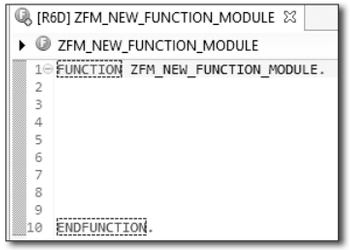


Figure 6.26 Newly Created Function Module in Eclipse

Function module parameters

Just like class methods, function modules have `IMPORTING`, `EXPORTING`, and `CHANGING` parameters. These are defined in the function module by entering the type of parameter followed by a declaration of the data type, as shown in Listing 6.21.

```
FUNCTION ZFM_NEW_FUNCTION_MODULE.
    IMPORTING
        ip_param TYPE i.
    EXPORTING
        ep_param TYPE i.
    CHANGING
        cp_param TYPE i.

ENDFUNCTION.
```

Listing 6.21 Adding Parameters to a Function Module

You can also pass table parameters using the `TABLES` keyword, but this is obsolete and unnecessary, because you can pass a parameter of a table type instead.

Creating Function Groups in Transaction SE80

If you are using Transaction SE80 as your ABAP IDE, select FUNCTION GROUP from the dropdown in the center of left side of the screen, type "ZFG_FUNCTION_GROUP" in the textbox below the dropdown in the center left side of the screen, and press Enter.

You will be prompted with a popup asking if you want to create ZFG_FUNCTION_GROUP because it doesn't exist (Figure 6.27). Click the YES Button.

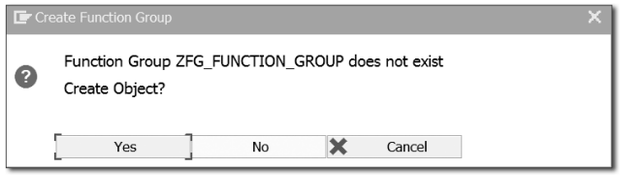


Figure 6.27 Create Function Group Popup

Creating a function group

You'll then be prompted with the CREATE FUNCTION GROUP popup pictured in Figure 6.28. Leave the function group name as ZFG_FUNCTION_GROUP, and enter "New function group" in the SHORT TEXT textbox. The SHORT TEXT and FUNCTION GROUP values are used by other developers trying to find your function group, so they should typically describe the type of data or objects that you're working with.

The PERSON RESPONSIBLE field will default to your username. You can leave this as is or change it, depending on your company's standards. Next, click the SAVE button to continue.

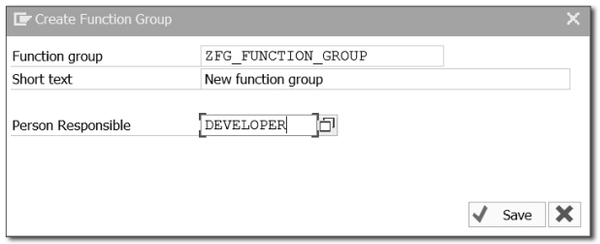


Figure 6.28 Create Function Group Popup

In the CREATE OBJECT DIRECTORY ENTRY popup, click the LOCAL OBJECT button or enter "$TMP" for the package name, and click the SAVE button. For production function groups, you should use a package created for your project.

Include files

Your function group has now been created containing two include files, which you should see listed on the left side of the screen (see

Figure 6.29). The `INCLUDE` files ending in `TOP` will contain any global data variables that will be accessible by any of the function modules in the function group. The `INCLUDE` ending in `UXX` will contain `INCLUDE`s for all of the function modules created.

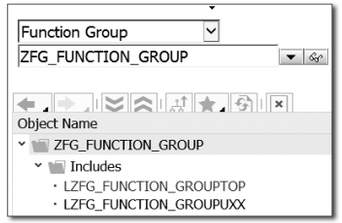


Figure 6.29 New Function Group Files in SE80

Creating a function module

To create the function module, you can either select FUNCTION MODULE from the dropdown, type "ZFM_NEW_FUNCTIONMODULE" in the textbox, and press Enter, or you can right-click the ZFG_FUNCTION_GROUP folder and select CREATE • FUNCTION GROUP.

You will then see the CREATE FUNCTION MODULE popup pictured in Figure 6.30. From the popup, enter "Z_NEW_FUNCTION_MODULE" for the function module name; unlike a global class method, it must be prefixed with Z. Because the code doesn't call the function group but instead calls the function module directly, the function module name is global, and each name can only be used once. A good practice is to prefix the function module name with something indicating the function group name.

Next, ensure that FUNCTION GROUP is set to the correct value; it should be ZFG_FUNCTION_GROUP for this example. Finally, to benefit other developers trying to find your function module, enter a description of what the function module does in the SHORT TEXT field; for this example, enter "Creating a new function module". Then, click SAVE.

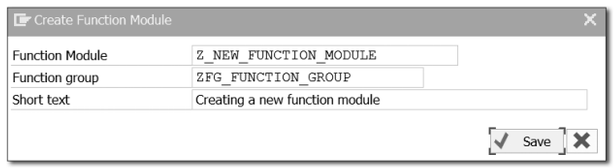


Figure 6.30 Create Function Module Popup

Function module parameters

You will now see a screen similar to the global class form-based view. Just like class methods, function modules have `IMPORTING`, `EXPORTING`, and `CHANGING` parameters. These are defined in the Import, Export, and Changing tabs. Function modules also have a Tables tab used to define parameters that are passed as tables, which is obsolete and should be avoided. Instead of using the tables parameters, you should pass a table type in one of the other parameters.

To add the `IMPORTING` parameter, select the Import tab and enter "IP_PARAM" for the parameter name, "TYPE" for Typing and "I" for Associated Type as shown in Figure 6.31. This will create an importing parameter of `IP_PARAM`.

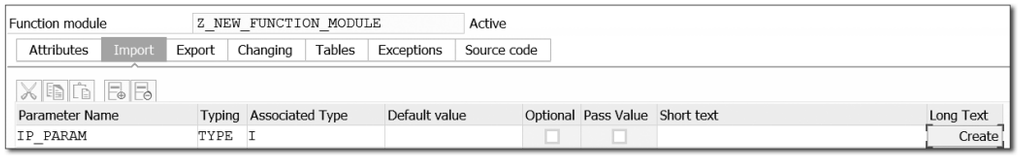


Figure 6.31 Adding an Import Parameter to a Function Module

Next, click on the Export tab and enter parameter "EP_PARAM" with Typing "TYPE" and Associated Type "I" as shown in Figure 6.32.

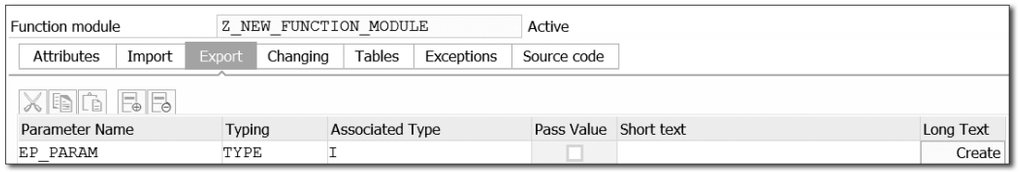


Figure 6.32 Adding an Export Parameter to a Function Module

Finally, select the Changing tab and enter parameter "CP_PARAM" with Typing "TYPE" and Associated Type "I".

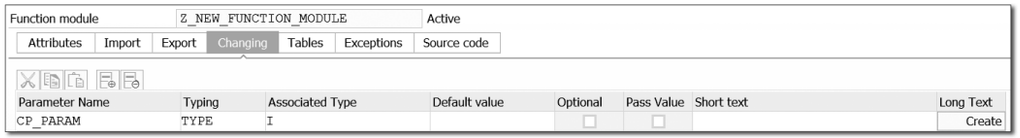


Figure 6.33 Adding a Changing Parameter to a Function Module

Now, you can select the Source Code tab, where you will be able to make changes to the function module, just as you made changes earlier to the method of a global class.

Calling Function Modules

You can call a function module using the `CALL FUNCTION` keyword followed by the function module name in single quotes (`'`). The parameters are passed by indicating the type of parameter and then the parameter name, equals sign, and the value that you want to pass to the parameter. Like we saw with class methods, a parameter importing into the function module is exporting from your code. The parameters importing into your program are always optional when calling a function module. An example calling `Z_NEW_FUNCTION_MODULE` is shown in Listing 6.22.

```
DATA: d_i TYPE i.
CALL FUNCTION 'Z_NEW_FUNCTION_MODULE'
    EXPORTING
        ip_param = 1
    IMPORTING
        ep_param = d_i
    CHANGING
        cp_param = d_i.
```

Listing 6.22 Calling a Function Module

Some function modules will return a parameter or table indicating the results of executing the function and any errors. Some function modules use exceptions to indicate whether or not the function call was successful. We will cover these types of exceptions in Chapter 10.

Form Subroutines

Form subroutines are probably the most commonly used way to modularize programs, even though they've been marked as obsolete in ABAP documentation.

Perform

Subroutines are easy to create. They use the `FORM` keyword followed by the subroutine name to define the beginning of the subroutine and `ENDFORM` to define the end of the subroutine. A subroutine can be called using the `PERFORM` keyword followed by the subroutine name and will execute any code within that subroutine, as shown in

Listing 6.23. No additional code can be entered after `ENDFORM`, just additional subroutines.

```
PERFORM my_subroutine.
FORM my_subroutine.
    WRITE: 'Hello World'.
ENDFORM.
```

Listing 6.23 Using a Subroutine

Declaring data in a subroutine

As with methods, local data can be declared within the subroutine and will only be accessible within that subroutine. Subroutines can also import parameters with the `USING` keyword and change parameters with the `CHANGING` parameter. These parameters work just like their function module and class-based counterparts. An example of a subroutine with parameters is shown in Listing 6.24.

```
DATA: d_i TYPE i.
PERFORM my_subroutine USING d_i
  CHANGING d_i.
  FORM my_subroutine
  USING up_param TYPE i
  CHANGING cp_param TYPE i.

ENDFORM.
```

Listing 6.24 Subroutine with USING and CHANGING Parameters

Summary

This chapter covered some modularization concepts, such as using separation of concerns to divide code into units that each complete only one function, and introduced object-oriented programming as a way to modularize code to meet the separation of concerns principle.

Remember that separation of concerns and object-oriented programming make your code easier to read and understand for the next developer who will have to look at it and fix any issues.

Next, you learned how to create classes and objects in ABAP. You discovered how methods can be used not only to modularize your code but also to make it compact through options such as recursive method calls.

We then covered global classes, which can be called by any program in the system, and how to create them both in Eclipse and using Transaction SE80 in SAP GUI.

The chapter concluded by looking at some obsolete code modularization techniques using function modules and subroutines. Even though you shouldn't be writing new subroutines and function modules, these are important to understand in order to maintain old ABAP systems.

In the next chapter, you will take everything you've learned so far in the book and apply it to create a new custom application that will run inside your ABAP system.

Contents

PART II Finishing Touches

Index

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

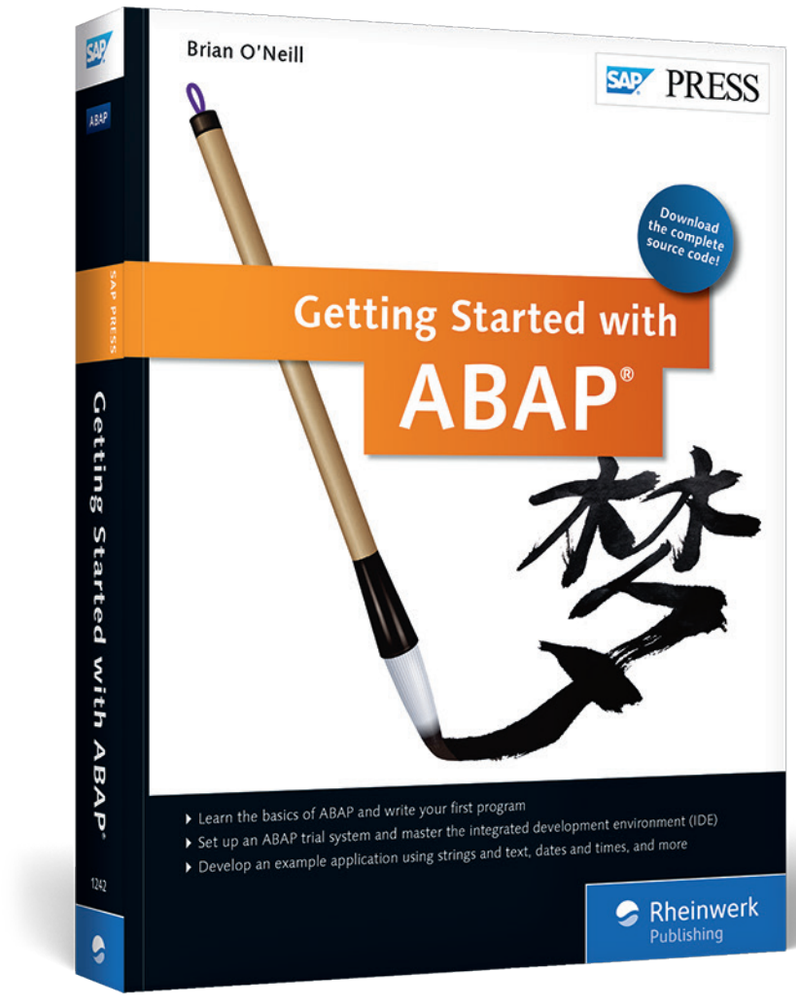
T

U

V

W

Z



Brian O'Neill is an ABAP developer with experience working across different SAP ERP modules and custom applications. He has worked in a variety of IT positions, from business analyst to ABAP developer. He is currently writing applications that can connect to an SAP backend using SAP Gateway. He holds a bachelor's degree in Computer Information Systems from California State University.

Brian O'Neill

Getting Started with ABAP

451 Pages, 2016, $49.95/€49.95
ISBN 978-1-4932-1242-2

www.sap-press.com/3869