SAP PRESS

Object-Oriented Programming with ABAP® Objects

- Make the move from procedural to object-oriented programming
- Learn to use encapsulation, inheritance, polymorphism, and more
- Work with ABAP Objects in the ABAP RESTful application programming model and the SAP BTP ABAP environment

3rd edition, updated and revised

Boggess · Hemond
Wood · Rupert

Rheinwerk Publishing

Contents

Preface 17

1 Introduction to Object-Oriented Programming 23

1.1 The Need for a Better Abstraction 23
1.1.1 The Evolution of Programming Languages 23
1.1.2 Moving Toward Objects 24

1.2 Classes and Objects 25
1.2.1 What Are Objects? 25
1.2.2 Introducing Classes 26
1.2.3 Defining a Class Interface 28

1.3 Establishing Boundaries 29
1.3.1 Encapsulation and Implementation Hiding 29
1.3.2 Understanding Visibility Sections 31

1.4 Reuse 31
1.4.1 Composition 31
1.4.2 Inheritance 32
1.4.3 Polymorphism 33

1.5 Object Management 34

1.6 UML Tutorial: Class Diagram Basics 35
1.6.1 What Are Class Diagrams? 36
1.6.2 Classes 37
1.6.3 Attributes 37
1.6.4 Operations 38
1.6.5 Associations 39
1.6.6 Notes 40

1.7 Summary 40

2 Working with Objects 41

2.1 Defining Classes 41
2.1.1 Creating a Class 42
2.1.2 Component Declarations 43
2.1.3 Implementing Methods 50

2.2 Working with Objects ... 51
2.2.1 Object References ... 52
2.2.2 Creating Objects ... 52
2.2.3 Object Reference Assignments ... 53
2.2.4 Accessing Instance Components ... 55
2.2.5 Accessing Class Components ... 58
2.2.6 Working with Events ... 60
2.2.7 Working with Functional Methods ... 64
2.2.8 Chaining Method Calls Together ... 67

2.3 Building Your First Object-Oriented Program ... 70
2.3.1 Creating the Report Program ... 70
2.3.2 Adding in the Local Class Definition ... 74

2.4 Working with Global Classes ... 76
2.4.1 Understanding the Class Pool Concept ... 76
2.4.2 Getting Started with the Class Builder Tool ... 76
2.4.3 Creating Global Classes ... 78
2.4.4 Using the Form-Based Editor ... 80
2.4.5 Using the Source Code Editor ... 88

2.5 Developing Classes Using ABAP Development Tools ... 88
2.5.1 What Is Eclipse? ... 89
2.5.2 Setting Up the ABAP Development Tools Environment ... 89
2.5.3 Working with the ABAP Development Tools Class Editor ... 93

2.6 Working with Constructor Expressions ... 101

2.7 UML Tutorial: Object Diagrams ... 104

2.8 Summary ... 106

3 Encapsulation and Implementation Hiding 107

3.1 Lessons Learned from Procedural Programming ... 107
3.1.1 Decomposing the Functional Decomposition Process ... 108
3.1.2 Case Study: A Procedural Code Library in ABAP ... 110
3.1.3 Moving Toward Objects ... 115

3.2 Data Abstraction with Classes ... 116

3.3 Defining Component Visibilities ... 117
3.3.1 Working with Visibility Sections ... 118
3.3.2 Understanding the Friend Concept ... 121

3.4 Hiding the Implementation ... 123

3.5 Designing by Contract 124
3.6 UML Tutorial: Sequence Diagrams 125
3.7 Summary 126

4 Object Initialization and Cleanup 129

4.1 Understanding the Object Creation Process 129
4.2 Working with Constructors 133
4.2.1 Defining Constructors 134
4.2.2 Understanding How Constructors Work 134
4.2.3 Class Constructors 136
4.3 Taking Control of the Instantiation Process 137
4.3.1 Controlling the Instantiation Context 138
4.3.2 Implementing the Singleton Pattern 139
4.3.3 Working with Factory Methods 141
4.4 Garbage Collection 143
4.5 Tuning Performance 144
4.5.1 Design Considerations 144
4.5.2 Lazy Initialization 144
4.5.3 Reusing Objects 145
4.5.4 Making Use of Class Attributes 146
4.6 UML Tutorial: State Machine Diagrams 146
4.7 Summary 147

5 Inheritance and Composition 149

5.1 Generalization and Specialization 149
5.1.1 Inheritance Defined 150
5.1.2 Defining Inheritance Relationships in ABAP Objects 151
5.1.3 Working with Subclasses 156
5.1.4 Inheritance as a Living Relationship 157
5.2 Inheriting Components 159
5.2.1 Designing the Inheritance Interface 159
5.2.2 Visibility of Instance Components in Subclasses 161
5.2.3 Visibility of Class Components in Subclasses 162
5.2.4 Redefining Methods 162

5.2.5 Instance Constructors ... 165
5.2.6 Class Constructors ... 166

5.3 The Abstract and Final Keywords ... 166
5.3.1 Abstract Classes and Methods ... 166
5.3.2 Final Classes ... 170
5.3.3 Final Methods ... 172

5.4 Inheritance Versus Composition ... 173

5.5 Working with ABAP Refactoring Tools ... 176

5.6 UML Tutorial: Advanced Class Diagrams, Part I ... 179
5.6.1 Generalizations ... 179
5.6.2 Dependencies and Composition ... 179
5.6.3 Abstract Classes and Methods ... 180

5.7 Summary ... 182

6 Polymorphism 183

6.1 Object Reference Assignments Revisited ... 183
6.1.1 Static and Dynamic Types ... 184
6.1.2 Casting ... 186

6.2 Dynamic Method Call Binding ... 189

6.3 Interfaces ... 191
6.3.1 Interface Inheritance Versus Implementation Inheritance ... 192
6.3.2 Defining Interfaces ... 193
6.3.3 Implementing Interfaces ... 197
6.3.4 Working with Interfaces ... 200
6.3.5 Nesting Interfaces ... 203
6.3.6 When to Use Interfaces ... 205

6.4 UML Tutorial: Advanced Class Diagrams, Part II ... 207
6.4.1 Interfaces ... 207
6.4.2 Providing and Required Relationships with Interfaces ... 208
6.4.3 Static Attributes and Methods ... 209

6.5 Summary ... 209

7 Component-Based Design Concepts 211

7.1 Understanding SAP's Component Model ... 211

7.2 The Package Concept ... 214
7.2.1 Why Do You Need Packages? ... 214
7.2.2 Introducing Packages ... 215
7.2.3 Creating Packages Using the Package Builder ... 217
7.2.4 Embedding Packages ... 224
7.2.5 Defining Package Interfaces ... 225
7.2.6 Creating Use Accesses ... 228
7.2.7 Performing Package Checks ... 229
7.2.8 Restriction of Client Packages ... 231
7.3 Package Design Concepts ... 233
7.4 UML Tutorial: Package Diagrams ... 235
7.5 Summary ... 237

8 Error Handling with Exception Classes 239

8.1 Lessons Learned from Prior Approaches ... 239
8.1.1 Lesson 1: Exception Handling Logic Gets in the Way ... 239
8.1.2 Lesson 2: Exception Handling Requires Varying Amounts of Data ... 240
8.1.3 Lesson 3: The Need for Transparency ... 241
8.2 The Class-Based Exception Handling Concept ... 241
8.3 Creating Exception Classes ... 243
8.3.1 Understanding Exception Class Types ... 243
8.3.2 Local Exception Classes ... 244
8.3.3 Global Exception Classes ... 245
8.3.4 Defining Exception Texts ... 247
8.3.5 Mapping Exception Texts to Message Classes ... 249
8.4 Dealing with Exceptions ... 250
8.4.1 Handling Exceptions ... 250
8.4.2 Cleaning Up the Mess ... 254
8.5 Raising and Forwarding Exceptions ... 255
8.5.1 System-Driven Exceptions ... 255
8.5.2 Raising Exceptions Programmatically ... 256
8.5.3 Propagating Exceptions ... 260
8.5.4 Resumable Exceptions ... 263
8.6 UML Tutorial: Activity Diagrams ... 267
8.7 Summary ... 269

9 Unit Tests with ABAP Unit 271

9.1 ABAP Unit Overview 271
9.1.1 Unit Testing Terminology 272
9.1.2 Understanding How ABAP Unit Works 272
9.1.3 ABAP Unit and Production Code 274

9.2 Creating Unit Test Classes 274
9.2.1 Unit Test Naming Conventions 274
9.2.2 Generating Test Classes for Global Classes 275
9.2.3 Defining Test Attributes 276
9.2.4 Building Test Methods 277
9.2.5 Working with Fixtures 278
9.2.6 Working with Test Seams 278
9.2.7 Defining Reusable Test Classes 280

9.3 Assertions in ABAP Unit 280
9.3.1 Creating and Evaluating Custom Constraints 281
9.3.2 Applying Multiple Constraints 282

9.4 Managing Dependencies 283
9.4.1 Dependency Injection 283
9.4.2 Private Dependency Injection 284
9.4.3 Partially Implemented Interfaces 284
9.4.4 Working with Test Doubles 285
9.4.5 Other Sources of Information 287

9.5 Case Study: Creating a Unit Test in ABAP Unit 288

9.6 Executing Unit Tests 291
9.6.1 Integration with the ABAP Workbench 291
9.6.2 Creating Favorites in the ABAP Unit Test Browser 292
9.6.3 Integration with the Code Inspector 293

9.7 Evaluating Unit Test Results 294

9.8 Measuring Code Coverage 294

9.9 Moving Toward Test-Driven Development 296

9.10 Behavior-Driven Development 296

9.11 UML Tutorial: Use Case Diagrams 297
9.11.1 Use Case Terminology 297
9.11.2 An Example Use Case 298
9.11.3 The Use Case Diagram 299
9.11.4 Use Cases for Requirements Verification 300
9.11.5 Use Cases and Testing 301

9.12 Summary 301

10 Business Object Development with BOPF 303

10.1 What Is BOPF? 303

10.2 Anatomy of a Business Object 305
10.2.1 Nodes 306
10.2.2 Actions 310
10.2.3 Determinations 312
10.2.4 Validations 313
10.2.5 Associations 315
10.2.6 Queries 318

10.3 Working with the BOPF Client API 319
10.3.1 API Overview 320
10.3.2 Creating Business Object Instances and Node Rows 323
10.3.3 Searching for Business Object Instances 326
10.3.4 Updating and Deleting Business Object Node Rows 327
10.3.5 Executing Actions 328
10.3.6 Working with the Transaction Manager 329

10.4 Where to Go from Here 330
10.4.1 Looking at the Big Picture 330
10.4.2 Enhancing Business Objects 331

10.5 UML Tutorial: Advanced Sequence Diagrams 331
10.5.1 Creating and Deleting Objects 332
10.5.2 Depicting Control Logic with Interaction Frames 333

10.6 Summary 334

11 ABAP RESTful Application Programming Model 335

11.1 Introduction 335
11.1.1 What Is the ABAP RESTful Application Programming Model? 335
11.1.2 Relationship to SAP Gateway 337
11.1.3 Business Object Runtime 338
11.1.4 Behavior Definitions 340
11.1.5 Model Classes 342

11.2 CDS and SAP Gateway Service Bindings 347
11.2.1 CDS Views 347
11.2.2 Associations 348

11.3 Modeling Behavior Using Object-Oriented Techniques 350
11.3.1 Creating the Service 350

11.3.2 Object-Oriented Implementation ... 360

11.4 Summary ... 370

12 ABAP Cloud 371

12.1 Introduction to SAP Business Technology Platform ... 372
12.1.1 What Is SAP Business Technology Platform? ... 372
12.1.2 Core Capabilities of SAP Business Technology Platform ... 374
12.1.3 ABAP and SAP Business Technology Platform ... 376

12.2 ABAP Environment Overview ... 377
12.2.1 Core Concepts ... 378
12.2.2 Key Architectural Components ... 383

12.3 Setting Up Your Cloud Development Environment ... 386
12.3.1 Transitioning from On-Premise to Cloud Development ... 386
12.3.2 Creating Your SAP BTP ABAP Environment ... 388

12.4 Case Study: Implementing ABAP RESTful Application Programming Model Elements via ABAP Cloud ... 395
12.4.1 Recreation of ABAP RESTful Application Programming Model Objects ... 396
12.4.2 Publishing and Consuming the Service ... 397

12.5 Summary ... 397

13 Best Practices and Design Patterns 399

13.1 Object-Oriented Analysis and Design ... 399

13.2 Creational Patterns ... 401
13.2.1 When to Use Creational Patterns in ABAP ... 403
13.2.2 Singleton Pattern ... 404
13.2.3 Factory Method Pattern ... 405

13.3 Structural Patterns ... 408
13.3.1 When to Use Structural Patterns in ABAP ... 409
13.3.2 Adapter Pattern ... 411
13.3.3 Decorator Pattern ... 414

13.4 Behavioral Patterns ... 419
13.4.1 When to Use Behavioral Patterns in ABAP ... 421
13.4.2 Strategy Pattern ... 421

13.5 Summary ... 425

Appendices 427

A Installing the Eclipse IDE 427
B The Authors 433

Index 435

Chapter 2
Working with Objects

This chapter introduces you to some basic ABAP Objects syntax and the relevant development tools that you'll need to start building object-oriented programs in ABAP.

Object-oriented programming (OOP), like many abstract concepts, is best learned by example. Now that we've gotten the basic definitions out of the way in Chapter 1, we're ready to turn our attention toward more practical matters and look at some basic syntax and sample code using ABAP Objects.

Because the primary unit of development for object-oriented programs is the class, we'll spend quite a bit of time in this chapter exploring the syntax used to define classes and their internal components. Once you're up to speed with basic syntax rules, we'll take a look at the tools used to define and maintain classes.

2.1 Defining Classes

Classes in ABAP Objects are declared using the `CLASS` statement block. This statement block is a wrapper of sorts, grouping all relevant class component declarations into two distinct sections:

- **Declaration section**
  This section specifies all the components defined within the class including attributes, methods, and events.
- **Implementation section**
  This section provides implementations (i.e., the source code) for the methods defined within the declaration section.

In the following sections, we'll unpack the syntax used to build out these sections and fully specify class types. For the purposes of this introductory section, our focus will be on defining *local classes* (i.e., classes that are defined within ABAP report programs and function group includes). However, in Section 2.4, you'll learn that this same syntax applies to the definition of global class types as well. The primary difference in the case of global classes is that you have a form-based editor in the Class Builder tool that spares you from manually typing out some of the declaration syntax.

2.1.1 Creating a Class

To define a new class type, you must declare it within a CLASS...DEFINITION...ENDCLASS statement block as shown in Listing 2.1. This statement block makes up the aforementioned declaration section of the ABAP class definition. As we noted earlier, this section is used to declare the primary components that make up a class, such as attributes and methods.

```
CLASS {class_name} DEFINITION [class_options].
  PUBLIC SECTION.
    [components]
  PROTECTED SECTION.
    [components]
  PRIVATE SECTION.
    [components]
ENDCLASS.
```

Listing 2.1 ABAP Class Declaration Section Syntax

If you look closely at Listing 2.1, you can see that the components of a class definition are organized into three distinct *visibility sections*: the PUBLIC SECTION, the PROTECTED SECTION, and the PRIVATE SECTION. Each of these visibility sections is optional, so it's up to you as a developer to determine which components go where—a subject that we'll consider at length in Chapter 3.

Besides the definition of the components that makeup the class's interface, the next most important task in defining a class is coming up with a good and meaningful name for it. As trivial as it may sound, this task is often harder than it looks. Part of the challenge stems from the fact that ABAP only gives you 30 characters to work with. You must come up with a meaningful name that fits within the confines of the syntax shown in Listing 2.2.

```
[{Namespace}|{Prefix}]CL_{Meaningful_Name}
```

Listing 2.2 ABAP Class Naming Convention

Listing 2.3 shows how this class naming syntax is applied to the various class types that may exist within the ABAP Repository. You'll see many more examples of this naming convention at work as you progress through the book.

```
LCL_LOCAL_CLASS            "Local Customer Class
ZCL_GLOBAL_CLASS           "Global Customer Class
CL_ABAP_MATCHER            "SAP-Standard Class (no namespace)
/BOWDK/CL_STRING_UTILS     "3rd-Party Class w/Namespace Prefix
```

Listing 2.3 Class Naming Examples

2.1.2 Component Declarations

As you've seen, the structure and makeup of a class is determined by its component definitions. Therefore, in this section, you'll learn about the different component types that you can define within a class. Before you get started, though, you first need to understand how components are grouped from a scoping perspective. Within a class declaration, there are two different types of components:

- **Instance components**
  Instance components, as the name suggests, are components that define the state and behavior of individual object instances. For example, an `Employee` class might have an instance attribute called `id` that uniquely identifies an employee within a company. Each instance of the `Employee` class maintains its own copy of the `id` attribute, which has a distinct value. Instance methods operate on these instance attributes to manipulate the object's state and perform instance-specific tasks.
- **Class components**
  Class components, on the other hand, are defined at the class level, meaning class components are shared across *all* object instances. Such components can come in handy in situations where you want to share data or expose utility functions on a wider scale. For example, in the `Employee` class scenario, you might use a class attribute called `next_id` to keep track of the next available employee ID number. This value could be used as a primitive number range object to assign the `id` instance attribute for newly created `Employee` objects.

  In practice, most of the classes you define will contain few class components. After all, it's hard to establish identity at the object level if all the data and/or functionality resides in global class components. However, you'll see that class components come in handy in certain situations, such as dealing with complex object creation scenarios or finding a home for utility functions.

Static Components

Class components are sometimes referred to as *static components* since they are statically defined and maintained at the class level. This is especially the case in other object-oriented languages such as Java or C#.

Internal Namespaces

Regardless of where you decide to define your components, it's important to keep in mind that all component names within an ABAP Objects class belong to the same internal namespace. For example, it's not possible to define an attribute and a method using the same name—even if they belong to different visibility sections. In the sections that follow, you'll learn that the adoption of good naming conventions makes it easy to avoid such naming collisions.

Attributes

As you learned in Chapter 1, attributes are essentially variables defined internally within a class or object. Attributes are defined in the same way that variables are defined in other ABAP programming modules. The primary difference in the case of classes is that you're contending with different contexts.

To put these contexts into perspective, consider the `LCL_CUSTOMER` sample class in Listing 2.4. Within this class definition, we've defined three different types of attributes:

- **Instance attributes**
  To define the properties that are unique to a particular customer instance, we've created several instance attributes such as `mv_id`, `mv_customer_type`, `mv_name`, and `ms_address`. As you can see in Listing 2.4, these instance attributes are defined using the familiar `DATA` keyword. Here, you can choose from any valid ABAP data type including structure types, table types, reference types, or even other class types.
- **Class attributes**
  The `sv_next_id` attribute is an example of a class attribute. As you can see, the only real difference syntax-wise between class attributes and instance attributes is the use of the `CLASS-DATA` keyword in lieu of the typical `DATA` keyword.
- **Constants**
  In the `PUBLIC SECTION` of our customer class, we've also defined several constants to represent the different customer types modeled in our class: `CO_PERSON_TYPE` for individuals, `CO_ORG_TYPE` for organizations, and `CO_GROUP_TYPE` for customer groups. These constants are defined just like any other constant using the `CONSTANTS` keyword. However, in the case of class constants, what we're really talking about is a specialized case of a class/static attribute (one that can't be modified at runtime).

```
CLASS lcl_customer DEFINITION.
   PUBLIC SECTION.
      CONSTANTS: CO_PERSON_TYPE TYPE c VALUE '1',
                 CO_ORG_TYPE    TYPE c VALUE '2',
                 CO_GROUP_TYPE  TYPE c VALUE '3'.
   PRIVATE SECTION.
      DATA: mv_id TYPE i,
            mv_customer_type TYPE c,
            mv_name TYPE string,
            ms_address TYPE adrc.
      CLASS-DATA: sv_next_id TYPE i.
ENDCLASS.
```

Listing 2.4 Declaring Attributes Within a Class

Though the ABAP compiler will generally allow you to define attributes with whatever name you prefer, we strongly recommend that you adopt a naming convention that

makes it easier to identify the scope of a given attribute. Table 2.1 describes the naming convention that will be used within this book.

Attribute Type	Naming Convention	Description
Instance attributes	M{Type}_{Meaningful_Name} Examples: mv_id ms_address mt_contacts	Here, the M implies that you're defining a *member variable*. The {Type} designator helps you more easily determine whether you're dealing with elementary variables (V), structures (S), internal tables (T), and so on. Aside from these scoping details, the rest of the instance attribute name is freeform and should be defined in such a way that it conveys meaning.
Class (static) attributes	S{Type}_{Meaningful_Name} Examples: sv_next_id	This convention is almost identical to instance attributes. However instead of the M for member variable, static attributes are prefixed with an S to imply that the attribute belongs to the static context.
Constants	CO_{MEANINGFUL_NAME}	Constants are typically defined in all caps using the CO_ prefix.

Table 2.1 Naming Convention for Defining Attributes

Methods

Methods are defined using either the METHODS statement for instance methods or the CLASS-METHODS statement for class methods. The syntax for both statement types is shown in the syntax diagram in Listing 2.5. Here, you can see that a method definition consists of a method name, an optional parameter list, and an optional set of exceptions that might occur. For this introductory section, we'll focus on the first two parts of a method definition. We'll circle back and cover exceptions in Chapter 8.

```
{CLASS-}METHODS {method_name}
   [IMPORTING parameters]
   [EXPORTING parameters]
   [CHANGING  parameters]
   [RETURNING VALUE(parameter)]
   [{RAISING}|{EXCEPTIONS}...].
```

Listing 2.5 Method Definition Syntax

As you can see in Listing 2.5, the first thing you specify in a method definition is the method's name. Since methods define the behavior of classes, it's important that you

come up with meaningful names that intuitively describe the method's purpose. Normally, it makes sense to prefix a method name with a strong action verb that describes the type of operation being performed. The sample class in Listing 2.6 provides some examples of this convention.

```
CLASS lcl_date DEFINITION.
  PUBLIC SECTION.
    METHODS:
      add IMPORTING iv_days TYPE i,
      subtract IMPORTING iv_days TYPE i,
      get_day_of_week RETURNING VALUE(rv_day) TYPE string,
      ...
ENDCLASS.
```

Listing 2.6 Defining Meaningful Names for Methods

After you come up with meaningful names for your methods, your next objective is to determine what sort of parameters (if any) the methods will need to perform their tasks. If you look at the syntax diagram from Listing 2.5, you can see that there are four different types of parameters that can be defined within a method's parameter list. Table 2.2 describes each of these parameter types in detail.

Parameter Type	Description
Importing	Importing parameters define the input parameters for a method. The values of an importing parameter cannot be modified inside the method implementation.
Exporting	Exporting parameters represent the output parameters for a method.
Changing	Changing parameters are input/output parameters that allow you to update or modify data within a method.
Returning	Returning parameters are used to define *functional methods*. You'll learn more about this parameter type when we look at functional methods in Section 2.2.7.

Table 2.2 Parameter Types for Method Definitions

To distinguish between the various parameter types within a method definition, method parameters are normally prefixed according to the convention described in Table 2.3. Here, the `{Type}` designator is once again used to differentiate between elementary data types (`V`), structure types (`S`), table types (`T`), and so on.

Parameter Type	Naming Convention
Importing	I{Type}_{Parameter_Name}
Exporting	E{Type}_{Parameter_Name}
Changing	C{Type}_{Parameter_Name}
Returning	R{Type}_{Parameter_Name}

Table 2.3 Method Parameter Naming Conventions

Regardless of the parameter's type, the syntax for declaring a parameter `p1` is shown in the syntax diagram in Listing 2.7. As you can see, this syntax provides you with several configuration options for defining a parameter:

- The optional `VALUE` addition allows you to specify that a parameter will be passed by *value* instead of by reference. For more details on this concept, check out the upcoming text box.
- You can use the `TYPE` addition to specify the parameter's data type. The addition is used in this context in the exact same way it's used to define normal variables or form parameters.
- You can use the `OPTIONAL` addition to mark a parameter as *optional*. Such parameters can be omitted during method calls on the consumer side.
- You can use the `DEFAULT` addition to specify a default value for a given parameter (which makes the parameter optional from a consumer perspective). The caller of the method can override this value as needed.

```
{ p1 | VALUE(p1)} TYPE type [OPTIONAL | {DEFAULT def1}]
```

Listing 2.7 Formal Parameter Declaration Syntax

Pass-by-Value versus Pass-by-Reference

At runtime, whenever a method that contains parameters is invoked, the calling program will pass parameters by matching up *actual parameters* (e.g., local variables in the calling program and literal values) in the method call with the *formal parameters* declared in the method signature (see Figure 2.1). Here, parameters are passed in one of two ways: *by reference* (default behavior) or *by value*.

Pass-by-value semantics is enabled via the aforementioned `VALUE` addition. Performance-wise, pass-by-value implies that a *copy* of an actual parameter is created and passed to the method for consumption. As a result, changes made to value parameters inside the method only affect the copy; the contents of the variable used as the actual parameter are not disturbed in any way. This behavior is illustrated at the top of Figure 2.1 with the mapping of parameter `a`. Here, whenever the method is invoked, a copy of

variable x is made and assigned to parameter a. As you might expect, this kind of operation can become rather expensive when you start dealing with large data objects.

Reference parameters, on the other hand, contain a reference (or *pointer*) to the actual parameter used in the method call. Therefore, changes made to reference parameters *are* reflected in the calling program. In Figure 2.1, this is illustrated in the mapping of parameter b. Here, if you were to change the value of parameter b inside the method, the change would be reflected in variable y in the calling program.

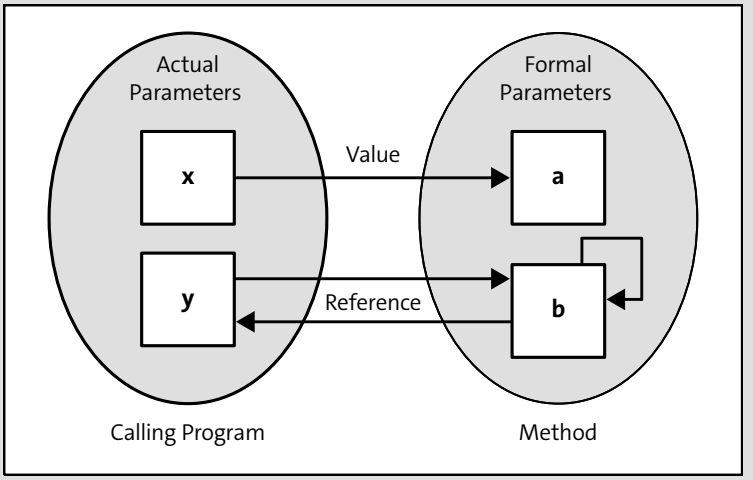


Figure 2.1 Mapping Actual Parameters to Formal Parameters

Since this behavior can potentially cause dangerous side effects, ABAP allows you to lock down reference parameters for editing inside methods by defining them as importing parameters. For example, if you define parameter b as an importing parameter, the compiler would complain if you try to modify its contents within the method body. In effect, importing parameters allow you to attain all the performance benefits of reference passing without the negative effects.

Collectively, a method's name and parameter list make up the method's *signature*. From the perspective of class consumers, method signatures determine the exact requirements for calling a particular method: which parameters to pass, the data types of the parameters being exchanged, and so on. As a method designer, it's important that you get these details right so that your methods are intuitive and easy to use. To that end, here are some design points to consider when defining method signatures:

- In general, keep the number of parameters being passed to or from methods to the bare minimum. You should assume that an object already has most of the information it needs (via its instance attributes) to perform a particular task, so you should only require a handful of parameters when defining a method.
- Define methods to perform one task. Avoid defining methods such as `copyDataAndWashCat()`.

- When performing generic operations where specific data types don't matter, incorporate the use of generic ABAP types so that the methods can be (re)used in a variety of contexts. For a list of available generic types, search for "generic ABAP types" in the ABAP Keyword Documentation.

Events

Besides the more common attributes and methods that you see in most object-oriented languages, ABAP Objects also allows you to define events that model certain types of occurrences within an object's lifecycle. Once again, you can distinguish between instance events that occur within a specific object instance and class events that are defined at the class level.

Listing 2.8 contains the basic syntax used to define instance events and class events. The parameters defined for an event are used to pass additional information about the event to interested event handler methods. Since this is a one-way data exchange, you're only allowed to define exporting parameters in an event definition. The syntax is pretty much identical to the syntax used to define exporting parameters in methods. The only difference in this case is that event parameters must be passed by value. Aside from the formally defined exporting parameters in an event definition, the system also supplies an implicit parameter called `sender` that contains a reference to the sending object (i.e., the object that raised the event).

```
EVENTS evt [EXPORTING parameters].
CLASS-EVENTS evt [EXPORTING parameters].
```

Listing 2.8 Event Declaration Syntax

Types

You can define custom data types within a class using the ABAP `TYPES` statement. These types are defined at the class level and are therefore not specific to any object instance. You can use these custom types to define local variables within methods, method parameter types, and so on. It's also possible to declare the use of global type pools defined within the ABAP Dictionary using the `TYPE-POOLS` statement.

The definition of class `LCL_PERSON` in Listing 2.9 demonstrates how types can be declared and used in a class definition. Here, we've defined a custom structure type called `TY_NAME` that's being used to define the person's `ms_name` attribute. The use of the `TY_` prefix in this case is by convention: Class-defined types are normally defined using the naming convention `TY_{Type_Name}`.

```
CLASS lcl_person DEFINITION.
   PRIVATE SECTION.
      TYPES: BEGIN OF ty_name,
                first_name TYPE char40,
                middle_initial TYPE char1,
```

```
              last_name TYPE char40,
            END OF ty_name.
      TYPE-POOLS: szadr.     "Business Address Services

      DATA: ms_name    TYPE ty_name,
            ms_address TYPE szadr_addr1_complete.
ENDCLASS.
```

Listing 2.9 Defining and Working with Class-Level Types

If you look closely at Listing 2.9, you can also see how type groups from the ABAP Dictionary are declared using the `TYPE-POOLS` statement. In this case, the class has declared the use of the `SZADR` type group from the *Business Address Services* package. Once this declaration is in place, you can use types such as the `SZADR_ADDR1_COMPLETE` type in attribute definitions and method signatures.

2.1.3 Implementing Methods

Any time you define methods within the declaration section of a class, you need to provide implementations for them in the implementation section. Such implementations are provided using `METHOD...ENDMETHOD` statement blocks that are nested inside of a `CLASS...IMPLEMENTATION...ENDCLASS` statement block, as shown in Listing 2.10.

```
CLASS lcl_date DEFINITION.
  ...
ENDCLASS.

CLASS lcl_date IMPLEMENTATION.
  METHOD add.
    mv_date = mv_date + iv_days.
  ENDMETHOD.

  METHOD subtract.
    mv_date = mv_date - iv_days.
  ENDMETHOD.

  METHOD get_day_of_week.
    "Implementation goes here..
  ENDMETHOD.
ENDCLASS.
```

Listing 2.10 Providing Implementations for Methods

As you can see in Listing 2.10, method implementations allow you to jump right into the code. There's no need to provide any further details about the method context, since you've already defined its signature in the declaration section. Within the

method processing block, you can implement the behavior of the class using regular ABAP statements in much the same way that you would implement subroutines and function modules from the procedural world. You'll see many examples of this in the sections that follow.

Syntax Restrictions

If you're coming to ABAP Objects from a procedural background, we should point out that there are a handful of ABAP language constructs that have been rendered obsolete/deprecated from within the object-oriented context. These changes came about as part of a language cleanup effort when SAP first introduced object-oriented extensions to ABAP. SAP saw an opportunity to do some internal housekeeping and ensure that deprecated language elements didn't make their way into new ABAP Objects classes.

For the most part, developers following current best practices shouldn't encounter these statements, as their use is generally frowned upon in any context. Still, if you're not sure which statements have become deprecated over the years, don't worry; the compiler will tell you if you've used one.

Before we wrap up our discussion on method implementations, let's take a moment to discuss *variable scoping rules* in an object-oriented context. Unlike procedural contexts, where the context is pretty cut-and-dry between global variables and local variables, method implementations get their hands on variables at several different scoping levels:

- Class attributes that behave like global variables
- Local variables whose scope is limited to the method that defines them
- Instance variables that sit somewhere in the middle, defining the state of a given object instance

With these additional options in play, you should be careful when qualifying variables so that their usage is clear. This makes the code more readable and prevents you from accidentally *hiding* instance or class attributes behind method-local variables with the same name. As you can expect, hiding instance or class attributes within a method can have some nasty side effects. Fortunately, if you stick to the naming conventions outlined in Section 2.1.2, this shouldn't ever be a concern.

2.2 Working with Objects

Now that you have a feel for how classes are defined in ABAP Objects, let's take a look at how these classes can be utilized from a consumer standpoint. In the sections that follow, you'll learn to define object reference variables, create new object instances, and put them to work in ABAP programs.

2.2.1 Object References

Before you begin creating new object instances, you first need to define variables to hold onto these objects so that you can address them within your programs. For reasons that will be explained in Chapter 4, the ABAP runtime environment does not allow direct access to an object in your programs. Instead, you're given indirect access to allocated objects via a special kind of variable called an *object reference variable*.

Listing 2.11 demonstrates the syntax used to define an object reference variable. Notice the use of the `TYPE REF TO` addition to indicate that `lo_date` is a reference variable. When reading this statement, keep in mind that the `lo_date` is an object reference variable that can point to objects of (class) type `LCL_DATE`.

```
DATA lo_date TYPE REF TO lcl_date.
```

Listing 2.11 Syntax to Define an Object Reference Variable

You can use this type of syntax to define object reference variables as instance attributes, local variables within method implementations, local variables within form routines, or even global variables.

2.2.2 Creating Objects

Once you define the appropriate object reference variables, you can begin creating object instances using the `CREATE OBJECT` statement shown in Listing 2.12. Functionally, this statement is processed behind the scenes as follows:

1. First, the ABAP runtime environment dynamically creates a new object of type `LCL_DATE`.
2. Then, after the object instance is created, control is handed off to a special method called a *constructor* that provides you with the ability to initialize the object instance before it's used. You'll learn more about constructor methods in Chapter 4.
3. Finally, once the object instance is initialized, the ABAP runtime environment fills in the `lo_date` variable with a reference that points to the newly created object.

```
DATA lo_date TYPE REF TO lcl_date.
CREATE OBJECT lo_date.
```

Listing 2.12 Instantiating an Object at Runtime

From a syntax perspective, that's all there is to instantiating objects. Anytime you want a new object reference, you simply use the `CREATE OBJECT` statement to allocate one on the fly. Of course, if you're not careful in maintaining your object reference variables, these objects can become orphaned. With that in mind, the next section focuses on the important topic of object reference assignments.

2.2.3 Object Reference Assignments

Since object reference variables are basically a special kind of variable, they can be used in assignment statements using the familiar equals (=) operator. Of course, when assigning object reference variables, it's important to remember *what* you're assigning. To put this concept into perspective, consider the assignment scenario in Listing 2.13.

```
DATA lo_date1 TYPE REF TO lcl_date.
DATA lo_date2 TYPE REF TO lcl_date.

CREATE OBJECT lo_date1.
CREATE OBJECT lo_date2.

lo_date1 = lo_date2.
```

Listing 2.13 Understanding Object Reference Assignments

Within the code excerpt in Listing 2.13, we have two object reference variables called `lo_date1` and `lo_date2` that point to newly created `LCL_DATE` objects. Prior to the assignment statement at the bottom of the code excerpt, the variable assignments resemble what is shown in Figure 2.2. Notice that the objects themselves are not stored within the object reference variables. Instead, the object reference variables contain an address for where the object exists in memory.

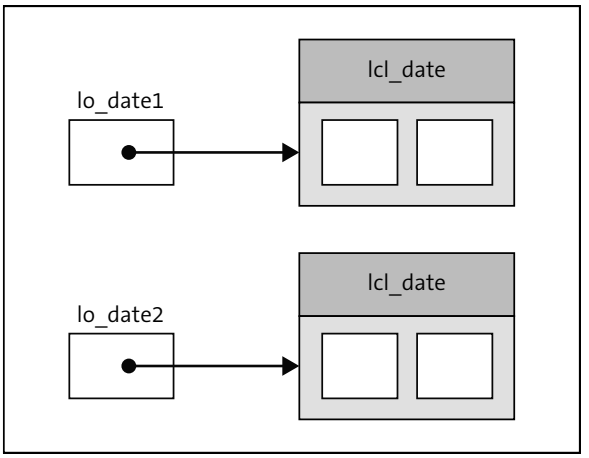


Figure 2.2 Understanding Object-Reference Assignments (Part 1)

The diagram in Figure 2.3 illustrates what things look like *after* the object reference assignment is performed at the bottom of Listing 2.13. Here, you can see that the assignment statement has copied the address of the `LCL_DATE` object instance pointed to by the `lo_date2` object reference into `lo_date1`. Now, both `lo_date1` and `lo_date2` point to the same object instance (i.e., the instance at the bottom of Figure 2.3).

If you look closely at the before and after memory snapshots in Figure 2.2 and Figure 2.3, you can draw several important conclusions about object reference assignments:

1. First, it should be clear that object reference assignments only copy the *addresses* of objects, and not the objects themselves. This implies that object reference assignments are relatively inexpensive from a performance standpoint.
2. Second, any time you have two or more object reference variables that point to the same object instance, changes made to the object via one object reference variable will be reflected in the other object reference variables. This should come as no surprise, since the object reference variables all point to the same object instance.
3. Finally, if an object instance is no longer pointed to by any live object reference variables, the object instance becomes *orphaned* and is no longer accessible from a programming context. In Chapter 4, you'll see how a special service of the ABAP runtime environment called the *garbage collector* cleans up these orphaned objects to recoup unused memory.

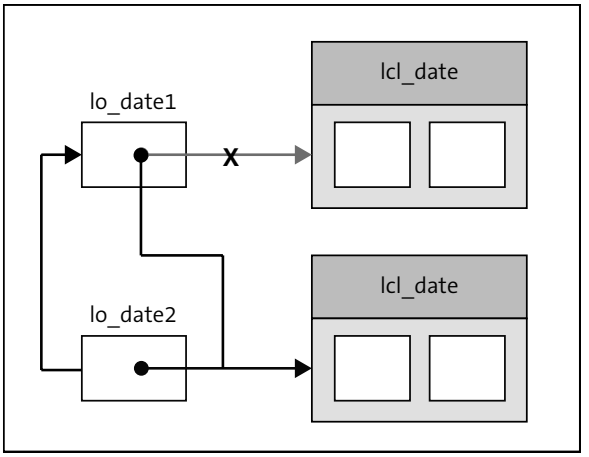


Figure 2.3 Understanding Object-Reference Assignments (Part 2)

With time and a little bit of practice, these concepts should become second nature. In the meantime, though, we recommend taking a methodical approach to creating object instances and performing object reference assignments. For example, consider the code excerpt in Listing 2.14. The intent was to create 10 date objects but, since there's only one object reference variable, the first 9 date objects are created and then subsequently orphaned.

```
DATA lo_date TYPE REF TO lcl_date.
DO 10 TIMES.
  CREATE OBJECT lo_date.
ENDDO.
```

Listing 2.14 An Invalid Idiom for Creating a Collection of Objects

Listing 2.15 corrects the error from Listing 2.14 by introducing an internal table of object references. Now, each new date object that's created is stored in a separate object reference variable within the table. As obvious as this may seem, these are the types of issues that can occur if you aren't careful with your object handling.

```
DATA lt_dates TYPE STANDARD TABLE OF REF TO lcl_date.
FIELD-SYMBOLS <lo_date> LIKE LINE OF lt_dates.

DO 10 TIMES.
  APPEND INITIAL LINE TO lt_dates ASSIGNING <lo_date>.
  CREATE OBJECT <lo_date>.
ENDDO.
```

Listing 2.15 Defining a Collection of Objects

Thinking (Object)ively

For many ABAP developers, the notion of reference variables is a foreign concept. If you find yourself getting tripped up by all this indirection, perhaps an analogy will help. Consider the relationship between a remote control and a TV set.

Remote controls are small, lightweight devices that make it easy for you to control a TV set. As long as you have your remote control, you can turn on the TV, change the channel, and control the volume as desired. However, if you were to lose the remote, then you'd no longer be able to access the TV (at least, not without getting off the couch). To guard against such occurrences, you could buy a universal remote as a backup. That way, you could program the universal remote to point to the TV's remote frequency. Once the universal remote is programmed, you could control the TV with either remote, since they both effectively point to the same TV.

Relating this back to our object reference discussion, object reference variables are rather like remote controls. As long as an object reference variable points to a particular object instance, you can use the object reference to control the object it points to. However, if you reassign the object reference or clear it out using the ABAP `CLEAR` statement, then you can no longer use it to access the object instance. This doesn't mean the object is deleted, the same way that your TV would still exist even if you lost your remote control. What it does mean is that you may no longer be able to access the object if you don't have another object reference variable on hand that happens to point to that object. This is the OOP equivalent of losing all of your remotes in the couch cushions.

The moral of the story is to treat object references with care. Make sure that you're really done with an object before clearing its object reference variables.

2.2.4 Accessing Instance Components

As you learned in the previous sections, object reference variables provide a handle for addressing object instances. Using this handle, you can access the instance components

of an object by building compound expressions using the object component selector operator (->), as shown in Listing 2.16. Here, you can see that the object component selector allows you to specify which instance component you want to access within a given object instance.

```
oref->attribute
oref->method()
CALL METHOD oref->method( )
```

Listing 2.16 Working with the Object Component Selector (Part 1)

What's the Proper Syntax for Calling a Method?

As you can see in Listing 2.16, there are two different ways to call methods. These days, the direct `oref->method()` option is the preferred option, as it more closely resembles syntax used in other object-oriented languages. The `CALL METHOD` statement is still valid, of course, but you should avoid it as a rule. In Section 2.2.7 and Section 2.2.8, you'll understand why it's a good idea to get into the habit of calling methods directly.

To demonstrate the use of the object component selector operator, let's take a look at an example. Imagine that you're modeling a 2D graphics system and want to create an object to represent points in the Cartesian coordinate system. If it's been a while since your last high school geometry class, a Cartesian coordinate system (or plane) is a two-dimensional grid that contains a horizontal x-axis and vertical y-axis (see Figure 2.4). To plot points on the graph, all you have to do is specify an x-coordinate and a y-coordinate. This is demonstrated in Figure 2.4 where we've plotted a point at (1,2).

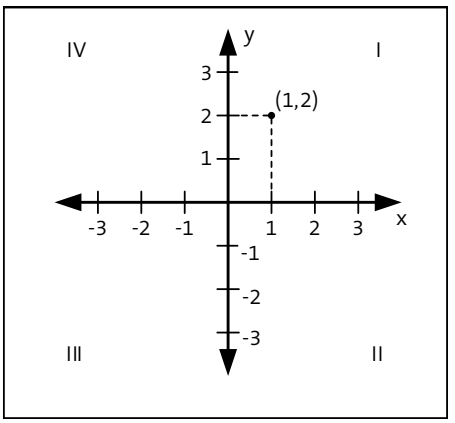


Figure 2.4 Modeling a Point Object in the Cartesian Coordinate System

To model our point object, we'll create a new class called `LCL_POINT`, as shown in Listing 2.17. This class contains three instance components: two instance attributes called `mv_x` and `mv_y` to represent the x and y coordinates, respectively, and an instance method

called get_distance() that can be used to calculate the Euclidian distance between the current point and some other point within the plane.

```
CLASS lcl_point DEFINITION.
   PUBLIC SECTION.
      DATA: mv_x TYPE p DECIMALS 2   "X-Coordinate
            mv_y TYPE p DECIMALS 2.  "Y-Coordinate

      METHODS get_distance IMPORTING io_point2
                           TYPE REF TO lcl_point
                           RETURNING VALUE(rv_distance) TYPE f.
ENDCLASS.

CLASS lcl_point IMPLEMENTATION.
   METHOD get_distance.
      DATA: lv_dx TYPE f,         "Diff. X
            lv_dy TYPE f.         "Diff. Y

      "Calculate the Euclidean distance between the points:
      lv_dx = io_point2->mv_x - me->mv_x.
      lv_dy = io_point2->mv_y - me->mv_y.

      rv_distance =
         sqrt( ( lv_dx * lv_dx ) + ( lv_dy * lv_dy ) ).
   ENDMETHOD.
ENDCLASS.
```

Listing 2.17 Working with the Object Component Selector (Part 2)

If you look closely at the implementation of the get_distance() method, you can see that the object component selector is used to access the instance attributes of two different objects: the io_point2 object passed to the method and the current point object. In the latter case, we're referring to the current point object's instance attributes using the me self-reference variable.

Where Does the me Self-Reference Variable Come From?

The me self-reference variable is a special instance attribute implicitly defined by the ABAP runtime environment whenever an object instance is created. As its name implies, the me reference variable points back to its containing object. If you've worked with other object-oriented languages such as Java, you can think of the me reference variable as being equivalent to the this self-reference variable.

From a usage perspective, the me self-reference variable can be used just like any other object reference variable. For example, in Listing 2.17 we used me to access the mv_x and mv_y instance attributes of the LCL_POINT class. Technically speaking, we didn't have to

use me to access these attributes. Instead, we could have simply referenced the attributes directly and the system would have quietly resolved the reference behind the scenes. The advantage of qualifying these references directly is that we make our intentions clear to the reader.

Another place where the me reference variable is used is in situations where you want to pass the current object instance as a parameter to another method. In this case, me provides you with a convenient mechanism for accessing the current object directly within a method implementation.

The code excerpt in Listing 2.18 demonstrates how to use the object component selector to access attributes and methods from outside of a class. Here, we're using the selector to:

- Initialize the instance attributes of a pair of point objects (lo_point_a and lo_point_b, respectively).
- Invoke the get_distance() method to calculate the distance between the two points.

```
DATA: lo_point_a  TYPE REF TO lcl_point,
      lo_point_b  TYPE REF TO lcl_point,
      lv_distance TYPE f.

"Instantiate both of the point objects:
CREATE OBJECT lo_point_a.
lo_point_a->mv_x = 1.
lo_point_a->mv_y = 1.

CREATE OBJECT lo_point_b.
lo_point_b->mv_x = 3.
lo_point_b->mv_y = 3.

"Calculate the distance & display the results:
lv_distance = lo_point_a->get_distance( lo_point_b ).
WRITE: 'Distance between point a and point b is: ',
       lv_distance.
```

Listing 2.18 Working with the Object Component Selector (Part 3)

2.2.5 Accessing Class Components

To demonstrate how to access class components, Listing 2.19 shows how we can enhance the LCL_POINT class we developed in the previous section to incorporate class components. Here, we'll introduce a new class method called create_from_polar() that can be used to create point objects using polar coordinates. To drive the conversion routine, we've also created a constant called CO_PI to represent the value of pi.

```
CLASS lcl_point DEFINITION.
  PUBLIC SECTION.
    CONSTANTS CO_PI TYPE f VALUE '3.14159265'.

    CLASS-METHODS:
      create_from_polar IMPORTING iv_r TYPE f iv_theta TYPE p
                        RETURNING VALUE(ro_point) TYPE REF TO lcl_point.
    ...
ENDCLASS.

CLASS lcl_point IMPLEMENTATION.
  METHOD create_from_polar.
    "Convert the angle measure to radians:
    DATA lv_theta_rad TYPE f.
    lv_theta_rad = ( iv_theta * CO_PI ) / 180.

    "Create a new point object and calculate the
    "X & Y coordinates:
    CREATE OBJECT ro_point.

    ro_point->mv_x = iv_r * cos( lv_theta_rad ).
    ro_point->mv_y = iv_r * sin( lv_theta_rad ).
  ENDMETHOD.
  ...
ENDCLASS.
```

Listing 2.19 Defining Class Components

Since our new `create_from_polar()` method is defined at the class level, we don't require an object reference to access it. Instead, we can access it via the static/class context using the class component selector operator (`=>`), as shown in Listing 2.20. Here, you can see how we're also accessing the `CO_PI` constant using the same kind of syntax: `{class_name}=>{class_component}`.

```
DATA lo_point TYPE REF TO lcl_point.
DATA lv_message TYPE string.

lo_point = lcl_point=>create_from_polar( iv_r = '3.6' iv_theta = '56.31' ).
lv_message =
  |Coordinates: ({ lo_point->mv_x }, { lo_point->mv_y })|.
WRITE: / lv_message.
lv_message = |PI is { lcl_point=>CO_PI }|.
WRITE: / lv_message.
```

Listing 2.20 Working with the Class Component Selector

If you look back at Listing 2.19, you'll notice that we didn't qualify the use of the `CO_PI` constant within the `create_from_polar()` method. Within the class itself, such qualifications are optional, since the class context is implicitly known. Whether or not you choose to formally qualify such references is strictly a matter of preference.

2.2.6 Working with Events

For developers coming into ABAP Objects with a background in other object-oriented languages such as Java or C#, the concept of events as first-class citizens of class definitions may seem a bit foreign. However, once you understand how events work, it's easy to see how they relate to common object synchronization patterns employed in those environments (e.g., the observer pattern).

From a conceptual perspective, events are a special kind of component you can use to model important milestones that might occur during an object's lifecycle. Such milestones could be unique to a particular object instance (instance events) or to the class itself (class events). In either case, whenever a particular milestone is reached, you can highlight the occurrence by *raising* an event. You can have interested parties (i.e., other objects) listen for these events by registering them as *event handlers*. This allows the ABAP runtime environment to automatically notify objects of the event.

This exchange is illustrated by the diagram in Figure 2.5. Here, we have a class called `LCL_PUBLISHER` that defines an instance event called `MESSAGE_ADDED`.

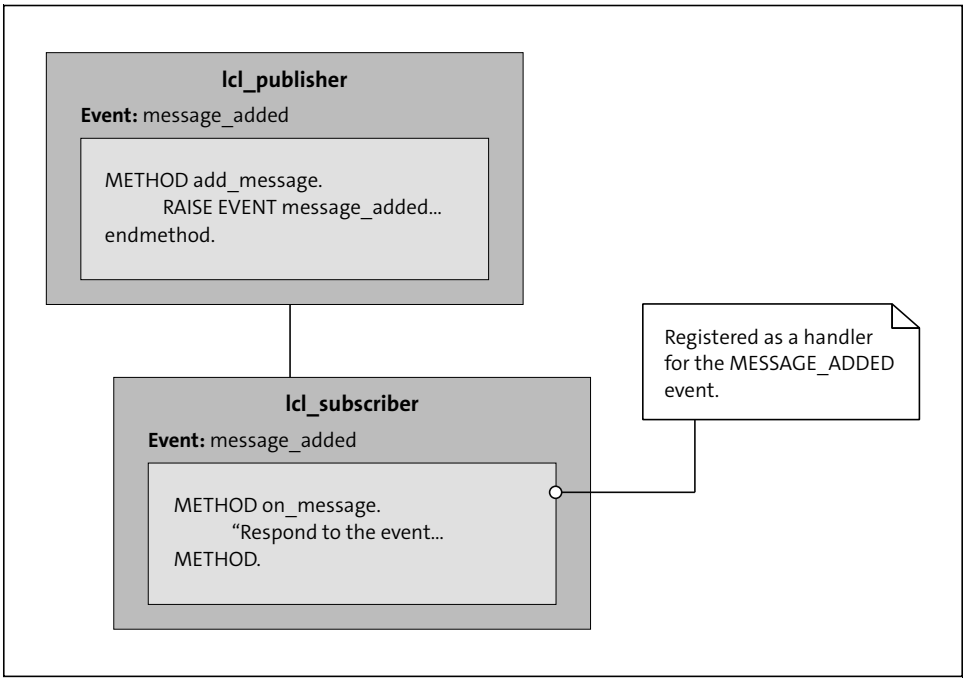


Figure 2.5 Understanding the Event Process Flow

This event is triggered whenever the publisher receives a new message via its add_message() instance method. On the other end of the exchange, we have another class called LCL_SUBSCRIBER that is registered as a *listener* for the MESSAGE_ADDED event. Now, whenever a new message arrives at the publisher, instances of LCL_SUBSCRIBER will be notified via the on_message() event handler method.

Event-Related Syntax

If you're looking at the event process flow in Figure 2.5, you might be wondering how the on_message() method was fired in response to the MESSAGE_ADDED event. Unlike the methods you've seen thus far, the on_message() method is defined as an *event handler method*. As you might expect, event handler methods are specialized methods that register themselves as listeners for particular types of events. You can define event handler methods within the same class that declared the event or in a completely separate class.

To declare event handler methods, you must once again enlist the aid of the METHODS statement, as shown in Listing 2.21. Here, the FOR EVENT...OF CLASS addition links the method with the corresponding event it's defined to handle. As you would expect, the importing parameter list must match the exporting parameter list defined by the event it's listening for.

```
METHODS {method_name}
  FOR EVENT {event} of CLASS {class_name}
  [IMPORTING p1 p2 ... [sender]].
```

Listing 2.21 Declaring Event Handler Methods

Once an event handler method is defined, you can register it to listen for events using the SET HANDLER statement, whose syntax is shown in Listing 2.22. Here, the handler tokens refer to event handler methods (without quotes) that are defined within the class from which the SET HANDLER statement emanates. The remaining additions are defined as follows:

- When registering event handler methods for instance events, you have a couple of options for identifying the scope of the event binding:
  - The optional FOR oref addition is used to bind an event handler method to a specific object instance.
  - Alternatively, you can use the ALL INSTANCES addition to bind an event handler method to all object instances.
- When registering event handler methods for class events, you don't have to specify an object context, so neither of the FOR oref and ALL INSTANCES additions apply.
- Finally, for both instance and class event bindings, you have the option of activating and deactivating an event registration using the ACTIVATION addition. Here, you can activate an event handler method using the true ('X') value or deactivate the method using the false (space) value.

```
SET HANDLER handler1 handler2 ... [FOR oref|{ALL INSTANCES}]
                                  [ACTIVATION {'X'|' '}].
```

Listing 2.22 Registering Event Handler Methods

The final piece to the event syntax puzzle is the `RAISE EVENT` statement, whose syntax is shown in Listing 2.23. As you can see, the syntax is straightforward: You simply specify the event being raised and pass along any parameters that event handlers will use to process the event downstream.

```
RAISE EVENT evt [EXPORTING p1 = a1 p2 = a2 ...].
```

Listing 2.23 Syntax for Raising Events

Putting It All Together

To see how all this comes together in real-life ABAP code, let's look at how you might build the `LCL_PUBLISHER` and `LCL_SUBSCRIBER` classes depicted in Figure 2.5. Listing 2.24 shows the code for the `LCL_PUBLISHER` class. Here, you can see how we've defined the `message_added` event using the `EVENTS` keyword introduced in Section 2.1.2. This event is then triggered from within the `add_message()` method using the `RAISE EVENT` statement.

```
CLASS lcl_publisher DEFINITION.
  PUBLIC SECTION.
    METHODS:
      add_message IMPORTING iv_message TYPE string,
      confirm_receipt IMPORTING iv_subscriber TYPE string.

    EVENTS:
      message_added
        EXPORTING VALUE(ev_message) TYPE string.
ENDCLASS.

CLASS lcl_publisher IMPLEMENTATION.
  METHOD add_message.
    DATA lv_message TYPE string.
    lv_message = |Publishing message: [{ iv_message }].|.
    WRITE: / lv_message.

    RAISE EVENT message_added
      EXPORTING
        ev_message = iv_message.
  ENDMETHOD.

  METHOD confirm_receipt.
    DATA lv_message TYPE string.
    lv_message = |Message processed by { iv_subscriber }.|.
```

```
    WRITE: / lv_message.
  ENDMETHOD.
ENDCLASS.
```

Listing 2.24 Defining and Raising Events

Listing 2.25 contains the definition of the LCL_SUBSCRIBER class, which is listening for messages issued from the LCL_PUBLISHER class. We've defined an event handler method called on_message() that will be used to process publication events at runtime. The event binding takes place within the constructor() method using the SET HANDLER statement. You'll learn more about constructor methods in Chapter 4, but for now, know that this method is invoked automatically whenever an LCL_SUBSCRIBER instance is created.

```
CLASS lcl_subscriber DEFINITION.
  PUBLIC SECTION.
    METHODS:
      constructor,

      on_message FOR EVENT message_added
                   OF lcl_publisher
                 IMPORTING
                   ev_message sender.

ENDCLASS.

CLASS lcl_subscriber IMPLEMENTATION.
  METHOD constructor.
    SET HANDLER on_message FOR ALL INSTANCES.
  ENDMETHOD.

  METHOD on_message.
    DATA lv_message TYPE string.
    lv_message = |Received message [{ ev_message }]|.
    WRITE: / lv_message.

    sender->confirm_receipt( 'LCL_SUBSCRIBER' ).
  ENDMETHOD.
ENDCLASS.
```

Listing 2.25 Defining and Registering an Event Handler Method

With both classes in place, you can run a test by passing a message to the add_message() method of the LCL_PUBLISHER class (see Listing 2.26). This will trigger the MESSAGE_ADDED event and allow you to see how the LCL_SUBSCRIBER class responds. Once you play around with this and learn how to interact with the event processing loop, you'll find that this feature offers many interesting possibilities.

```
DATA lo_publisher TYPE REF TO lcl_publisher.
DATA lo_subscriber TYPE REF TO lcl_subscriber.

CREATE OBJECT lo_publisher.
CREATE OBJECT lo_subscriber.

lo_publisher->add_message( 'Ping...' ).
```

Listing 2.26 Testing the Event Processing Loop

2.2.7 Working with Functional Methods

As we've stated from the outset, one of the main goals with OOP is to develop code that's intuitive and easy to read. One of the ways that object-oriented languages achieve this is by providing a syntax that resembles the sentence structure of spoken languages. For example, if you think about a method call, you have a subject (either an object or a class) and a verb (the method being called). With a little bit of creativity and proper naming, you can build statements that even people without a technical background can read and understand (at least conceptually).

To make your code flow even better, you can employ the use of *functional methods*. As the name suggests, functional methods are used to compute a single discrete value. The value in this approach is that you can plug in functional methods in the operand positions of various ABAP statements to build powerful expressions.

Listing 2.27 illustrates the basic syntax used to declare functional methods. Here, as before, you can declare `IMPORTING` parameters to provide inputs to the method. The lone output of the method is provided in the form of the `RETURNING` value parameter. As is the case with other parameter types, you're generally free to define the type of the returning parameter using the same rules that apply for `EXPORTING` parameters. However, type selection does play a role in determining whether a functional method can be used as an operand in selected ABAP statements.

```
METHODS func_method
   [IMPORTING parameters]
   RETURNING VALUE(rval) TYPE type
   [EXCEPTIONS...].
```

Listing 2.27 Functional Method Declaration Syntax

To demonstrate how functional methods are used in ABAP code, let's look at an example. In Listing 2.28, we've created a string tokenizer class called `LCL_STRING_TOKENIZER` that can be used to parse through delimited records and make it easy to access individual string tokens. This class defines two functional methods:

- The `has_more_tokens()` method is a Boolean method that can be used to determine whether there are more tokens in the sequence.

- The `next_token()` method provides a simple mechanism for accessing the next token in the sequence.

```
CLASS lcl_string_tokenizer DEFINITION.
  PUBLIC SECTION.
    METHODS:
      constructor IMPORTING iv_string TYPE csequence
                            iv_delimiter TYPE csequence,

      has_more_tokens RETURNING VALUE(rv_result) TYPE abap_bool,

      next_token RETURNING VALUE(rv_token) TYPE string.

  PRIVATE SECTION.
    DATA mt_tokens TYPE string_table.
    DATA mv_index TYPE i.
ENDCLASS.

CLASS lcl_string_tokenizer IMPLEMENTATION.
  METHOD constructor.
    SPLIT iv_string AT iv_delimiter INTO TABLE me->mt_tokens.

    IF lines( me->mt_tokens ) GT 0.
      me->mv_index = 1.
    ELSE.
      me->mv_index = 0.
    ENDIF.
  ENDMETHOD.

  METHOD has_more_tokens.
    IF me->mv_index LE lines( me->mt_tokens ).
      rv_result = abap_true.
    ELSE.
      rv_result = abap_false.
    ENDIF.
  ENDMETHOD.

  METHOD next_token.
    READ TABLE me->mt_tokens INDEX me->mv_index INTO rv_token.
    ADD 1 TO me->mv_index.
  ENDMETHOD.
ENDCLASS.
```

Listing 2.28 Working with Functional Methods (Part 1)

The code excerpt in Listing 2.29 demonstrates how you can use your string tokenizer class within regular ABAP code. Notice how we're using the `has_more_rows()` method as the basis of the logical expression that drives the `WHILE` loop that processes the string tokens. At runtime, this method will be invoked prior to the evaluation of the logical expression, and the returned value will be used to determine if the `WHILE` loop should continue. Not only does this save you a few lines of code, it also makes the code much more intuitive.

```
DATA lo_tokenizer TYPE REF TO lcl_string_tokenizer.
DATA lv_token TYPE string.

CREATE OBJECT lo_tokenizer
  EXPORTING
    iv_string = '09/13/2005'
    iv_delimiter = '/'.

WHILE lo_tokenizer->has_more_tokens( ) EQ abap_true.
  lv_token = lo_tokenizer->next_token( ).
  WRITE: / lv_token.
ENDWHILE.
```

Listing 2.29 Working with Functional Methods (Part 2)

Table 2.4 provides some further examples of places where you can use functional methods in common ABAP expressions.

ABAP Expression	Where It's Used

Conditional expressions (e.g., `IF` and `WHILE` statements)	As an operand in a logical expression.
	Example:
	`IF oref->get_weight( ) GT 100.`
	`  ...`
	`ENDIF.`

`CASE`	As an operand in a logical expression.
	Example:
	`CASE oref->get_type( ).`
	`WHEN oref->get_value1( ).`
	`  ...`
	`ENDCASE.`

Table 2.4 Using Functional Methods in Expressions

ABAP Expression	Where It's Used

LOOP AT, DELETE, and MODIFY	As part of the logical expression in a WHERE clause.
	Example:
	`LOOP AT itab`
	`  WHERE field EQ oref->get_val( ).`
	`  ...`
	`ENDLOOP.`

Table 2.4 Using Functional Methods in Expressions (Cont.)

Enhancements to Functional Methods

A functional method's signature supports exporting and changing parameters in addition to the singular returning parameter. Such methods can still be used inline within regular ABAP expressions; the extra exporting and changing parameters simply come along for the ride.

Predicative method calls are another useful feature. As the name suggests, these are functional method calls where the result is used as a predicate in logical expressions. To put this into perspective, consider the way that we're using predicative method calls to refactor the WHILE loop in Listing 2.29. Notice that we no longer have to compare the result of the has_more_tokens() method using a logic expression. In this context, if the returning value parameter of has_more_tokens() is initial, then the result is false; all non-initial values evaluate to true. Therefore, we can define the signature of methods using Boolean approximation types such as ABAP_BOOL or pretty much any other data type. Of course, for readability's sake, we encourage you to define your functional methods using familiar Boolean types wherever possible.

```
WHILE lo_tokenizer->has_more_tokens( ).
  ...
ENDWHILE.
```

Note that no syntactical changes are required in the implementation of methods such as has_more_tokens() to exploit this functionality. You can continue to develop functional methods as per usual, only now you can incorporate them into functional expressions in a concise and readable manner.

2.2.8 Chaining Method Calls Together

Chained method calls are a type of syntactic sugar popular among ABAP developers. They make it easy to consolidate a handful of operations into a single line of code.

To understand how chained method calls work, consider this example. In Listing 2.30, we've created a simple string utilities class called LCL_STRING. Within this class, we've defined several functional methods that perform various operations on a string value:

converting the string to upper case, trimming of leading/trailing whitespace, and replacing characters. This is all standard fare, until you get to the part where each of these methods passes back a copy of the `me` self-reference variable. This subtle addition to the code is what makes method chaining possible.

```
CLASS lcl_string DEFINITION.
  PUBLIC SECTION.
    METHODS:
      constructor IMPORTING iv_string TYPE csequence,

      trim RETURNING VALUE(ro_string) TYPE REF TO lcl_string,

      upper RETURNING VALUE(ro_string)
                 TYPE REF TO lcl_string,

      replace IMPORTING iv_pattern TYPE string
                        iv_replace TYPE string
              RETURNING VALUE(ro_string)
                   TYPE REF TO lcl_string,

      get_value RETURNING VALUE(rv_value) TYPE string.

  PRIVATE SECTION.
    DATA mv_string TYPE string.
ENDCLASS.

CLASS lcl_string IMPLEMENTATION.
  METHOD constructor.
    me->mv_string = iv_string.
  ENDMETHOD.

  METHOD trim.
    me->mv_string =
      condense( val = me->mv_string from = `` ).
    ro_string = me.
  ENDMETHOD.

  METHOD upper.
    me->mv_string = to_upper( val = me->mv_string ).
    ro_string = me.
  ENDMETHOD.

  METHOD replace.
    REPLACE ALL OCCURRENCES OF REGEX iv_pattern
         IN me->mv_string WITH iv_replace.
```

```
    ro_string = me.
  ENDMETHOD.

  METHOD get_value.
    rv_value = me->mv_string.
  ENDMETHOD.
ENDCLASS.
```

Listing 2.30 Working with Chained Methods (Part 1)

The code excerpt in Listing 2.31 demonstrates how to implement chained method calls from a code perspective. Here, you can see how we're taking an existing string and performing multiple operations on it in one go. This starts with the call to the `trim()` method. This method strips off the leading/trailing whitespace and then passes back a copy of the `me` self-reference. The resultant object reference is then used as the basis for the subsequent call to the `upper()` method, which follows the same kind of pattern. The call chain ultimately terminates with the call to `get_value()`, at which time we receive the formatted text "`PAIGE_A_PUMPKIN`".

Note

We added the line break between the calls to `upper()` and `replace()` so that the statement would fit onto a printed page in the book. Within the ABAP Editor, this sort of line break would result in a syntax error.

```
DATA lo_string TYPE REF TO lcl_string.
DATA lv_new_value TYPE string.

CREATE OBJECT lo_string
  EXPORTING
    iv_string = `  Paige A Pumpkin  `.

lv_new_value =
  lo_string->trim( )->upper( )->
    replace( iv_pattern = `\s` iv_replace = '_' )->get_value( ).

WRITE: / lv_new_value.
```

Listing 2.31 Working with Chained Methods (Part 2)

As you can see in the example, chained method calls make it easy to string together related operations in one condensed statement. For simple operations like the ones demonstrated in Listing 2.31, this makes logical sense. For more complex statements, though, chained method calls are probably a bad idea. We leave it to you as a responsible developer to know when it makes sense to sacrifice readability to save a few keystrokes.

2.3 Building Your First Object-Oriented Program

In the previous section, we looked at several examples that demonstrated how to work with objects. However, since these code excerpts were isolated, you might be wondering how all these pieces fit together in actual ABAP programs. With that in mind, this section will demonstrate the creation of a simple report program that utilizes a local class. As you'll come to find out, these concepts apply equally to the incorporation of local classes to function group definitions, module pool programs, and so on.

2.3.1 Creating the Report Program

To get things started, let's create the report program that will drive our demo. If you're new to ABAP development, this can be achieved by performing the following steps:

1. To begin, log onto the system and open the Object Navigator (Transaction SE80).
2. In the object list selection box in the **Repository Browser** on the left-hand side of the screen, choose the **Local Objects** list option (see Figure 2.6).

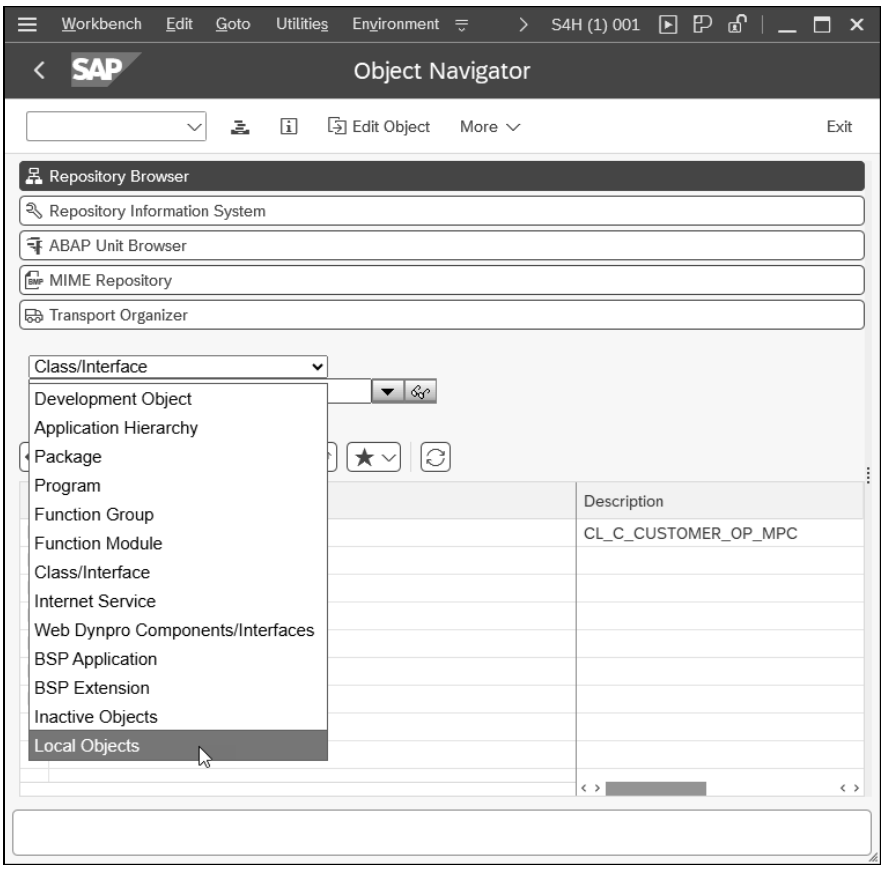


Figure 2.6 Selecting the Local Objects Repository View

3. This will pull up a tree view of locally defined development objects for your user account, as shown in Figure 2.7. To create a new report program, right-click on the top-level object node (i.e., **$TMP JWOOD** in Figure 2.7) and select **Create • Program** from the menu.

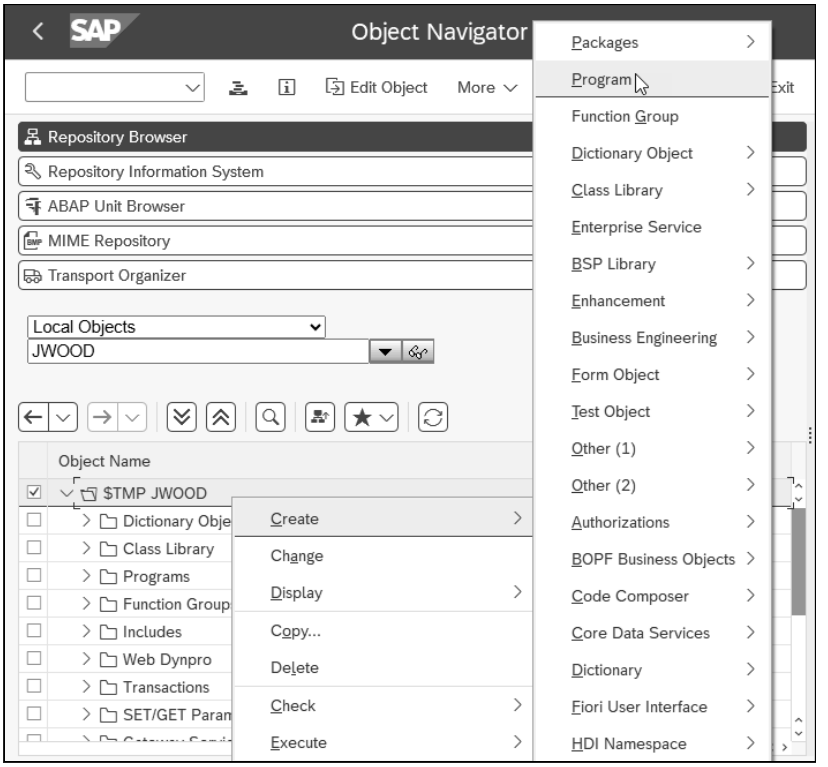


Figure 2.7 Creating a Report Program (Part 1)

4. Next, you'll be presented with the **Create Program** dialog box shown in Figure 2.8. At this step, simply specify the name of the report program (we called our report `YDATE_DEMO`) and press Enter to continue. Note that you shouldn't select the **Create with TOP Include** checkbox in this case since you're just building a simple report.

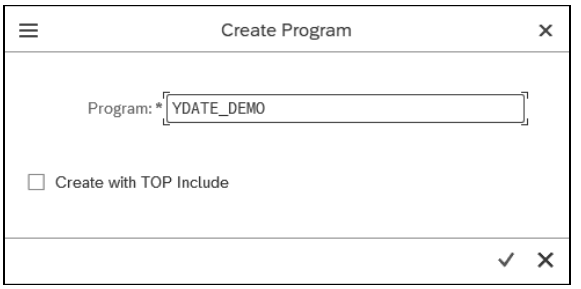


Figure 2.8 Creating a Report Program (Part 2)

5. Figure 2.9 shows the next dialog box in the creation process, where you provide a program title and additional attributes concerning the program setup. For the purpose of this simple demonstration, you can leave the default settings and click on the **Save** button to continue.

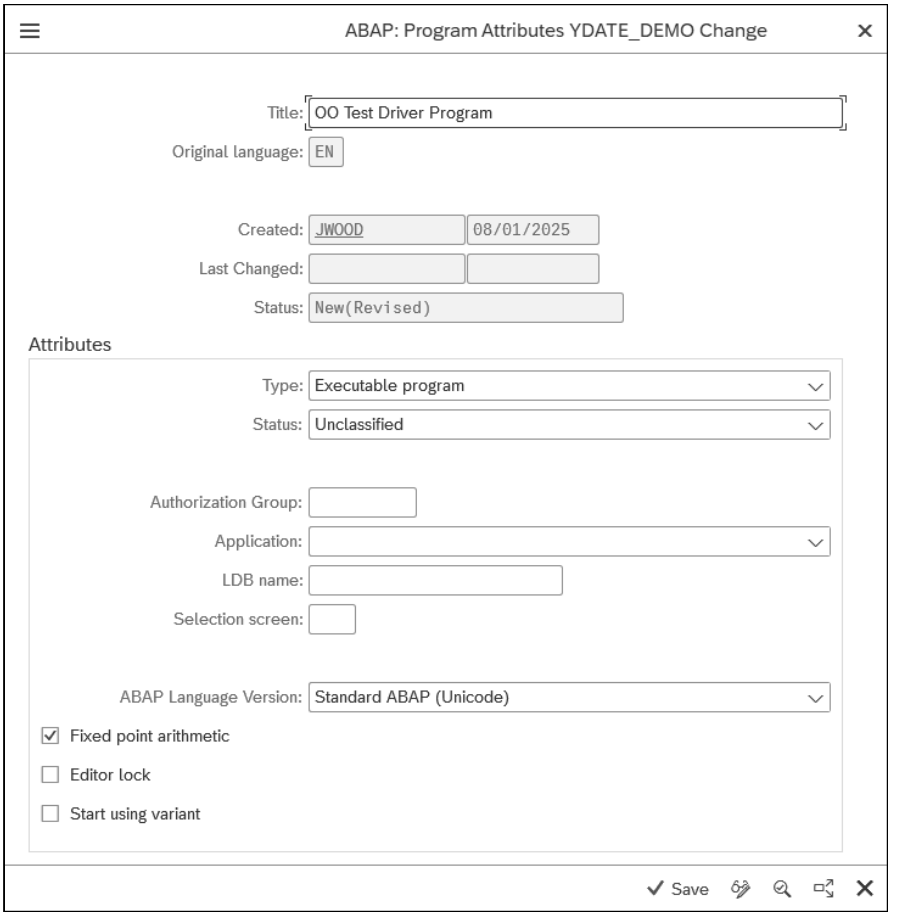


Figure 2.9 Creating a Report Program (Part 3)

6. At the next step, as shown in Figure 2.10, you'll be asked to select a package to store the object in within the ABAP Repository. Since this is a demo program, leave the default **$TMP** package selection and click the **Save** icon to continue. That way, the program will only be defined locally and can't be transported.
7. Finally, if all goes well, you should end up at an editor screen like the one shown in Figure 2.11. From there, you can get started with your coding exercise.

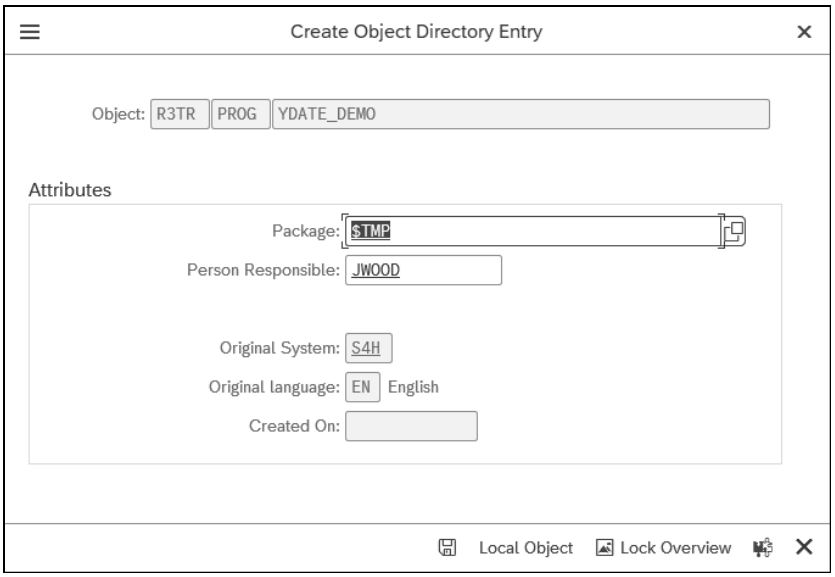


Figure 2.10 Creating a Report Program (Part 4)

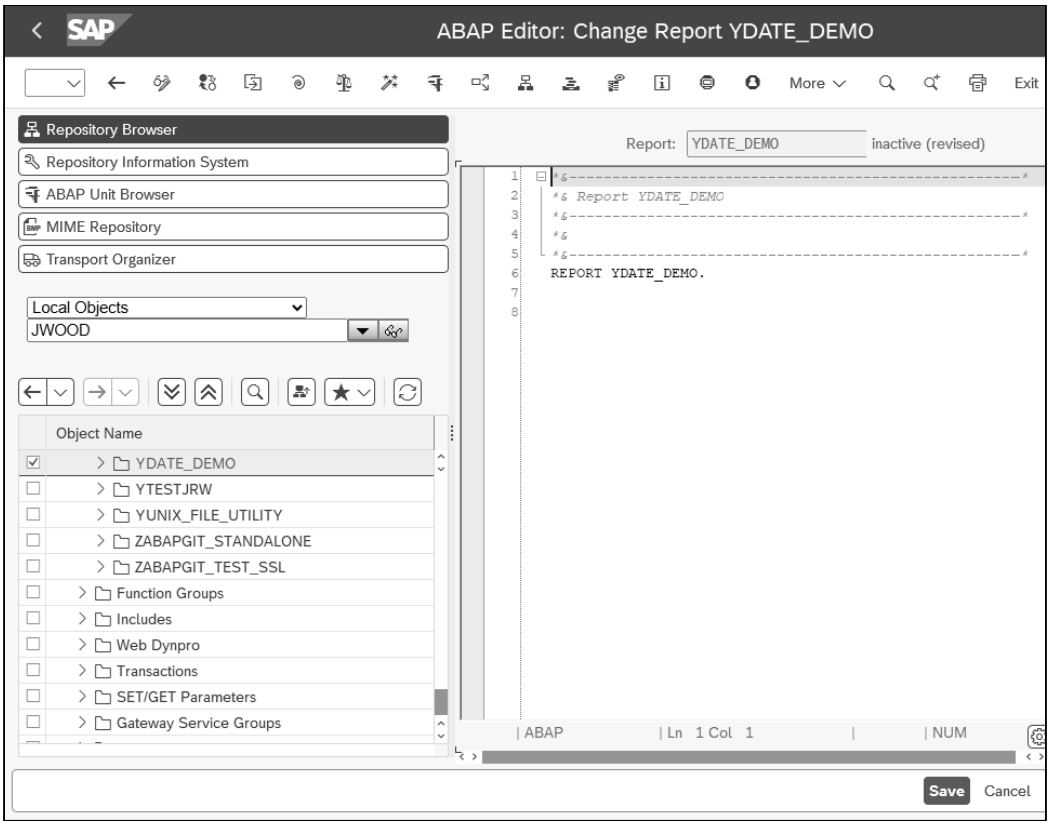


Figure 2.11 Creating a Report Program (Part 5)

2.3.2 Adding in the Local Class Definition

Once the report program is created, you can begin defining your local class in one of two ways:

- You can start keying in the class definition directly underneath the `REPORT` statement just like you would for other type definitions.
- Or, you can create an `INCLUDE` program and key in the class definition there.

The ABAP compiler doesn't care which option you choose, so it's up to you to decide how best to organize your code. For now, we'll keep things simple and define the class directly within the report program (see Listing 2.32). In Section 2.4, we'll take a closer look at some logistical implications when defining classes.

```
REPORT ydate_demo.
CLASS lcl_date DEFINITION.
  ...
ENDCLASS.

CLASS lcl_date IMPLEMENTATION.
  ...
ENDCLASS.
```

Listing 2.32 Defining Local Classes within a Report Program

Once the local class is defined, you can access it from within the report program in several different ways:

- You could define global object reference variables and then use those variables to create and use objects from within report events such as `START-OF-SELECTION` and `END-OF-SELECTION`.
- You could define local object reference variables within subroutines called from within the report program and access the objects that way.
- If the class defines a `main` class method, you could invoke that directly and let the class itself drive the main program logic.

Since this is a book about OOP, we tend to prefer the third option, as it frees us from having to mix-and-match programming paradigms. The code excerpt in Listing 2.33 demonstrates this approach. Here, you can see how the main program logic is driven by the `main()` class method, which is accessed directly within the `START-OF-SELECTION` event module. From here, it's OOP as per usual. Figure 2.12 shows what the program output looks like. If you want to try this out for yourself, you can download a complete version of the program from the book's source code bundle (available at *www.sap-press.com/6093*).

```
REPORT zoopbook_date_demo.
CLASS lcl_date DEFINITION.
  PUBLIC SECTION.
    CLASS-METHODS:
      main.
      ...
ENDCLASS.

CLASS lcl_date IMPLEMENTATION.
  METHOD main.
    DATA lo_birth_date TYPE REF TO lcl_date.
    DATA lv_message TYPE string.

    CREATE OBJECT lo_birth_date
      EXPORTING
        iv_date = '20030113'.

    lv_message =
     |Andersen was born on a
       { lo_birth_date->get_day_of_week( ) }.|.
    WRITE: / lv_message.

    lv_message =
      |Official birth date:
        { lo_birth_date->get_long_format( ) }.|.
    WRITE: / lv_message.
  ENDMETHOD.
ENDCLASS.

START-OF-SELECTION.
  lcl_date=>main( ).
```

Listing 2.33 Integrating Local Classes Inside Report Programs

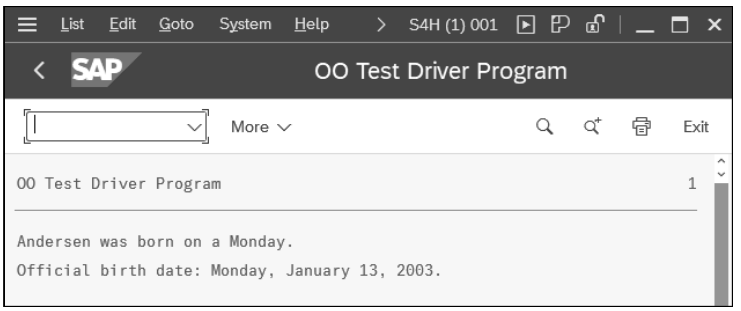
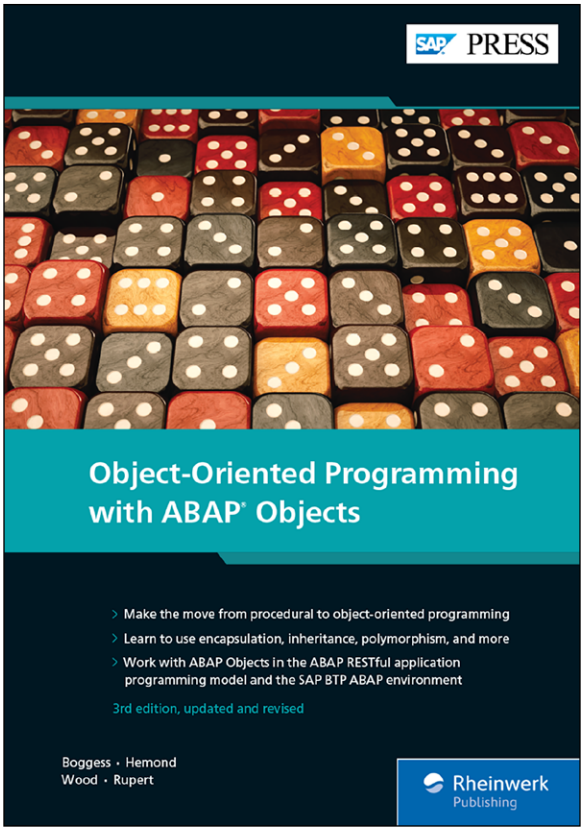


Figure 2.12 Output of the Example Program



Boggess, Hemond, Wood, Rupert

Object-Oriented Programming with ABAP® Objects

- Make the move from procedural to object-oriented programming
- Learn to use encapsulation, inheritance, polymorphism, and more
- Work with ABAP Objects in the ABAP RESTful application programming model and the SAP BTP ABAP environment

www.sap-press.com/6093

The Authors
Jeffrey Boggess, Colby Hemond, James Wood, and Joseph Rupert work at Bowdark Consulting. Their experience with SAP solutions includes ABAP programming, integration, cloud data architecture, and digital transformation.

ISBN 978-1-4932-2714-3 • 448 pages • 01/2026
E-book: $84.99 • **Print book:** $89.95 • **Bundle:** $99.99

Rheinwerk Publishing