Who should read this manual

This manual is for programmers with no previous experience using the O₂ system. It introduces the basic O₂ and O₂C concepts needed to begin using and programming with O₂.

Other documents available are outlined, click below.

See O₂ Documentation set.
This manual is divided into the following chapters:

- 1 - Introduction
- 2 - Objects, Classes and Types
- 3 - Programming Basics
- 4 - O₂ Tutorial
# 1 Introduction

## 1.1 System overview

- What is an object? ................................................................. 10
- Object uniformity ................................................................. 11
- Standard software ................................................................ 11

## 1.2 O2 Product Line

- O2Engine ............................................................................... 14
- O2Engine API ................................................................------- 15
- Programming languages and O2 .......................................... 16
- O2C ..................................................................................... 16
- OQL ..................................................................................... 17
- O2Look ............................................................................... 18
- O2Graph ............................................................................. 19
- O2Tools ............................................................................... 20
- O2Kit .................................................................................. 20

## 1.3 Documentation ................................................................ 21

## 1.4 Customer services and training ...................................... 21

## 1.5 Manual overview ............................................................. 22

# 2 Objects, Classes and Types

## 2.1 Objects and Values ........................................................... 24

- Basic structure ........................................................................ 24

## 2.2 Classes ............................................................................. 26

- Types ................................................................................... 27
- Methods ............................................................................... 31
- Class Specification and Visibility ....................................... 33
- Notions of Inheritance and Composition .......................... 35

## 2.3 Persistence ....................................................................... 36

- Named Objects ...................................................................... 36
- Named Values ...................................................................... 36

## 2.4 Chapter Summary .............................................................. 37
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>3</th>
<th>Programming Basics</th>
<th>39</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Inheritance</td>
<td>40</td>
</tr>
<tr>
<td>3.2</td>
<td>O2C</td>
<td>46</td>
</tr>
<tr>
<td>3.3</td>
<td>Database Management</td>
<td>60</td>
</tr>
<tr>
<td>3.4</td>
<td>The O2 System</td>
<td>63</td>
</tr>
<tr>
<td>3.5</td>
<td>Chapter Summary</td>
<td>68</td>
</tr>
<tr>
<td>4</td>
<td>O2 Tutorial</td>
<td>69</td>
</tr>
</tbody>
</table>

## 3 Programming Basics

### 3.1 Inheritance

- Class inheritance
- Redefinition
- Late binding
- Static binding

### 3.2 O2C

- Run body
- Objects: new and init
- Values: atomic and complex
- Object value: * operator
- Complex types: -> operator
- Lists
- Sets
- Bits and strings
- Object Sharing
- Copying values
- Generic Display tools

### 3.3 Database Management

- Persistence
- Transactions

### 3.4 The O2 System

- Schema
- Base
- Importing a schema
- Named System

### 3.5 Chapter Summary

## 4 O2 Tutorial

### 4.1 Creating an O2 system

### 4.2 Launching O2

### 4.3 Creating and populating the database

Create schema
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set schema</td>
<td>74</td>
</tr>
<tr>
<td>Create class</td>
<td>74</td>
</tr>
<tr>
<td>Named objects and values</td>
<td>77</td>
</tr>
<tr>
<td>Confirm the schema modification</td>
<td>78</td>
</tr>
<tr>
<td>Create base</td>
<td>78</td>
</tr>
<tr>
<td>Set base</td>
<td>78</td>
</tr>
<tr>
<td><strong>4.4 Beginning to program your database</strong></td>
<td>79</td>
</tr>
<tr>
<td>Add an object to the database</td>
<td>80</td>
</tr>
<tr>
<td><strong>4.5 Methods</strong></td>
<td>85</td>
</tr>
<tr>
<td>add_child</td>
<td>85</td>
</tr>
<tr>
<td>marriage</td>
<td>87</td>
</tr>
<tr>
<td>init</td>
<td>88</td>
</tr>
<tr>
<td>title</td>
<td>89</td>
</tr>
<tr>
<td><strong>4.6 Starting to query your database</strong></td>
<td>92</td>
</tr>
<tr>
<td><strong>4.7 Linking methods to applications</strong></td>
<td>94</td>
</tr>
<tr>
<td>Run application</td>
<td>95</td>
</tr>
<tr>
<td>Adding programs</td>
<td>96</td>
</tr>
</tbody>
</table>

| INDEX | 97 |
1 Introduction

Congratulations! You are now a user of the object-oriented database management system O₂.

O₂ is a revolutionary object-oriented database system that is particularly well suited for developing large-scale client/server applications in both business and technical software development.

This manual is aimed at programmers with no previous experience with O₂ or any other object-oriented language. It introduces the various concepts you need to know to begin using and programming with O₂.

This manual is in no way an exhaustive account of the O₂ system and should be used in conjunction with the special training courses run by O₂ and its associated distributors. Refer to all the other documentation available.

This introductory chapter is divided into the following sections:

- System overview - the basic concepts
- O₂ Product Line - what each product offers
- Manual overview.
1.1 System overview

The O₂ system is a distributed object-oriented database management system which can be accessed with various standard programming languages such as C, C++ and Java. In addition to its 4th generation object language, the O₂ system has an object SQL-like query language, a graphical user interface and a complete graphical programming environment.

The O₂ system is built around three basic concepts:
- The object - encapsulation of data and behavior.
- Object uniformity- one object from screen to disk.
- Standard software.

What is an object?

As the name suggests, the object-oriented database system is made up of objects as opposed to records you find in traditional database systems.

An object encapsulates data and behavior whereas a record simply contains data.

The object definition comprises both the data or value type and the behavior of the object or methods.

An O₂ object can consequently contain very complex data: alphanumeric, graphic, audio, image, video, etc.

Figure 1.1: Data and behavior
System overview

Object uniformity

In traditional database systems, data is managed differently depending on where it is in the system: disk data is managed using a DBMS, memory data is managed using a programming language and screen space using a user interface. The translation of data between these different formats is usually inefficient. In the O2 system this problem does not exist because objects are represented identically on disk, in memory and on screen as shown in Figure 1.2. As you manipulate objects in the same way anywhere in the system, you have a faster development time, easier maintenance and improved run time performance.

![Diagram showing object uniformity](image)

Figure 1.2: The same object from disk to screen, traditional system (left) and O2 (right).

Standard software

The final guiding principle behind the O2 system is that it is a totally standard system. The O2 system is written in C and C++, and runs under Unix. Its interface tools are written in C++ and run under X11 and OSF/ Motif. O2 is language independent, so you can develop applications in C, C++, Java and the 4GL O2C. In addition, you can utilize previously developed applications.
1.2 O₂ Product Line

The system architecture of O₂ is illustrated in Figure 1.3.

The O₂ system can be viewed as consisting of three components. The Database Engine provides all the features of a Database system and an object-oriented system. This engine is accessed with Development Tools, such as various programming languages, O₂ development tools and any standard development tool. Numerous External Interfaces are provided. All encompassing, O₂ is a versatile, portable, distributed, high-performance dynamic object-oriented database system.

Database Engine:

- O₂Store: The database management system provides low level facilities, through O₂Store API, to access and manage a database: disk volumes, files, records, indices and transactions.

- O₂Engine: The object database engine provides direct control of schemas, classes, objects and transactions, through O₂Engine API. It provides full text indexing and search capabilities with O₂Search and spatial indexing and retrieval capabilities with O₂Spatial. It includes a Notification manager for informing other clients connected to the same O₂ server that an event has occurred, a Version manager for handling multiple object versions and a Replication API for synchronizing multiple copies of an O₂ system.
O2 Product Line: Standard software

Programming Languages:
O2 objects may be created and managed using the following programming languages, utilizing all the features available with O2 (persistence, collection management, transaction management, OQL queries, etc.)

- C       O2 functions can be invoked by C programs.
- C++     ODMG compliant C++ binding.
- Java    ODMG compliant Java binding.
- O2C     A powerful and elegant object-oriented fourth generation language specialized for easy development of object database applications.
- OQL     ODMG standard, easy-to-use SQL-like object query language with special features for dealing with complex O2 objects and methods.

O2 Development Tools:

- O2Graph Create, modify and edit any type of object graph.
- O2Look Design and develop graphical user interfaces, provides interactive manipulation of complex and multimedia objects.
- O2Kit Library of predefined classes and methods for faster development of user applications.
- O2Tools Complete graphical programming environment to design and develop O2 database applications.

Standard Development Tools:

All standard programming languages can be used with standard environments (e.g. Visual C++, Sun Sparcworks).

External Interfaces:

- O2Corba Create an O2/Orbix server to access an O2 database with CORBA.
- O2DBAccess Connect O2 applications to relational databases on remote hosts and invoke SQL statements.
- O2ODBC Connect remote ODBC client applications to O2 databases.
- O2Web Create an O2 World Wide Web server to access an O2 database through the internet network.
O₂Engine

O₂Engine has all the features of a database engine providing transparent management of data persistence, data sharing and data reliability, as well as all the features of an object-oriented system including the manipulation of complex objects with identity, classes, types, methods, multiple inheritance, overriding and late binding of methods.

![Client/server architecture](image)

Figure 1.4: Client/server architecture

O₂Engine has a client/server architecture as shown in Figure 1.4. The server process provides persistence, disk management, concurrency control, data recovery and database security. The client process manipulates O₂ objects and values, and optimizes associative access.

In a given O₂ system you can have multiple bases and schemas that reside on one or more O₂ volumes distributed over a network.

Following the O₂ persistence model any object or value can be transient or persistent, O₂Engine allows you to dynamically change this status.
O2Engine is language independent and provides persistence by attachment. There are three possible types of interfaces:

- O2Engine API - a set of low level functions.
- Standard languages such as C and C++.
- The 4th generation language O2C.

O2Engine API

O2Engine API is an application programming interface that gives you access to O2Engine and offers direct control of data manipulation and therefore high performance for software integration.

O2Engine API is in fact a library of functions you can call from C or C++ applications or from any application written in a language supporting a C interface.

With direct access to the system, O2Engine API implements all the features of the O2 data model: complex objects, classes, indexes, transactions, object encapsulation, simple and multiple inheritance, and late binding of methods.
Programming languages and O₂

There are various languages with which you can program O₂: C, C++, Java and O₂C.

For all these languages, O₂ provides the same features:
- Persistence by reachability.
- Strong typing.
- Collection management.
- Access to all O₂ tools and all applications (C, C++, Java,...).

O₂C

O₂C, a superset of C, is a powerful, object-oriented 4th generation language that is specialized for developing object-oriented database applications.

O₂C is an extension of C with features for complex object manipulation and user interface generation. Its advantages include the possibility to run partially written code, reuse code in different schemas, write database queries, carry out transaction management and provide database consistency.
OQL

OQL is an object-oriented SQL-like query language. It can be used in two different ways, either as a function in a program or as a stand-alone query language.

OQL can be used as a function called from the various programming languages used with O₂. OQL allows you to manipulate complex values and methods. Each construct produces a result which can then be used in another query. In addition, methods can be triggered to modify the database. Overall, programming is easier because OQL can filter values.

OQL can also be used interactively as a stand-alone query language to simply query the database. Interactive features include fast and simple browsing of the database and integration with O₂ Look, the graphical user interface generator.

Figure 1.7: Examples of OQL queries
O₂Look

O₂Look enables you to design and create a graphical user interface for your object-oriented database applications. O₂Look is unique because it automatically creates the graphical display of any database object.

O₂Look offers software developers a powerful integrated tool enabling on-screen graphical display and interactive manipulation of complex and multimedia objects without any drawing or programming.

Some of its advantages include:
- Direct cut/ copy/ paste of objects.
- The application can control the interactive dialog between the application and the end user.
- O₂Look automatically creates a menu from which object methods can be triggered and which displays a dialog box for entering parameters.
- You can easily customize the O₂Look presentation to meet your specific needs.

Figure 1.8: O₂Look menus and dialog box
O₂Graph

O₂Graph is a specially designed module which enables you to create, modify, display and edit any kind of graph; and is fully integrated within O₂Look. It is ideal for applications in areas such as technical documentation, network management, software engineering, CIM, etc.

O₂Graph features include a set of reusable classes with a full set of methods so that you can manipulate graphs in the database, e.g. to create and remove nodes from a graph or insert links.

O₂Graph has a graphical editor which allows you to display and develop your graphs on-screen.

The following five display layouts are available:

- Coordinate.
- Grid.
- Tree.
- Dag.
- Graph.

![Figure 1.9: O₂Graph (tree layout)](image-url)
**O₂Tools**

O₂Tools is a complete graphical programming environment that greatly improves programmer efficiency and productivity, and consequently reduces development costs. It has the following features:

- Graphical browsers to visualize, conceptualize and implement objects in schemata and databases.
- Graphical editors to display and edit the schema and database.
- Source manager that ensures consistency of all application sources.
- All schema sources are stored in the database.
- Versions of method, program and function bodies.
- Automatic recompilation.
- Interface with OQL.

![Figure 1.10: O₂Tools](image)

**O₂Kit**

O₂Kit is a set of O₂ classes and methods you can import and customize, e.g. date and time, dialog boxes, text, color images, etc.
1.3 Documentation

This manual is a beginners manual. To get full details on how to use your O₂ system and all its various modules, you must refer to all the other O₂ manuals. For a description of all O₂ manuals see the beginning of this manual. (You will find the same description at the beginning of each individual manual.)

1.4 Customer services and training

For a full understanding of O₂ and O₂C, you must follow our specially designed training courses. For fuller details please contact our training department.

Other services to help you design, build and develop your applications are also available with O₂ Technology, including a Hotline and different Consulting Services.
1.5 Manual overview

This user manual is divided up into the following chapters:

- **Chapter 1 - Introduction**
  This is an introductory chapter that outlines the basic principles behind the O₂ system including the object, data integration and standard software, and then details each of the O₂ products.

- **Chapter 2 - Getting started**
  This chapter introduces the basic concepts you need to define your database structure in order to set up your database including objects and values, classes, methods, class inheritance and persistence.

- **Chapter 3 - Programming basics**
  This chapter details class inheritance and gives you what you need in O₂C to start programming. It then describes persistence and transaction management, and how to set up the O₂ system.

- **Chapter 4 - A tutorial**
  This a step-by-step tutorial, showing how to start the O₂ system, how to create and populate a simple database, how to write methods, how to link them into applications, and how to run queries. We do not attempt to be exhaustive nor present all possible programming options.
Objects, Classes, and Types

DEFINING YOUR APPLICATION
DATABASE STRUCTURE

Before beginning to actually program an O₂ application you must first define the structure of your database.

To do this you define the various classes and methods which determine the structure of the objects you are going to manage in the database.

This chapter introduces the basic concepts you need to define and set up your database. The following areas are covered:

- Objects and Values
- Classes including Types, Class Specification and Visibility, and Methods, and notions of Class Inheritance and Composition
- Persistence of named objects and values
- Chapter Summary

How the schema, bases and applications interact in the O₂ system is described in the next chapter.
2.1 Objects and Values

Information in O₂ is organized into objects and values.

An object is composed of an identifier, data and the behavior related to this data which includes the definition of operations upon the data and the implementation of these operations - an object is said to encapsulate data and its behavior.

Basic structure

An object has the following three features:

- **Object identity**
  An object has an identifier internal to the system and transparent to you the programmer. It is independent from the data contained in the object as well as from the relationships the object has with other objects. This enables data to be modified without modifying the identifier.

- **Value**
  Data contained in the object is the value of the object and the definition of the structure of this value is called the type.

- **Methods**
  The behavior related to the data or value is described by a set of program modules or methods. A data can only be modified through methods thereby closely binding values and behavior within an object.

For example,

![Object Diagram]

*Figure 2.11: O₂ object*

You can see that this organization greatly differs from traditional database systems which manage information in data records.
Objects and Values

Note

An object can contain far more complex information than a record.

In addition to objects containing simple data types such as numbers or strings, you can also have objects containing such complex data as tables, graphics, photos, voices, music and combinations of other objects.

- Class

Objects with the same value type and methods are grouped together into the same class.

An object is characterized by its class.

Figure 2.12: $O_2$ class
2.2 Classes

In O₂ you can have objects with the same type of data or value structure and using the same methods. Such objects with the same structure and behavior are grouped together into a class.

A class is taken to be the description of a group of related objects specifying the class type and methods that apply to all the objects of that class.

For example,

```plaintext
class Person
type tuple (name: tuple (last_name: string,
                 first_name: string),
        address: tuple (street: string,
                        city: City,
                        country: Country),
        photo: Bitmap,
        age: integer,
        read spouse: Person,
        read children: list (Person),
        public dossier_no: real)

method public is_adult: boolean,
        add_child (child: Person),
        ....
end
```

**Note**

The class name follows the convention of always consisting of an upper case letter followed by any number of upper or lower case letters, underscore characters or digits. Any other name consists of a lower case letter followed by any number of lower case letters or underscore characters.

This section now describes each part of the class description:

- **Types** - the different types available.
- **Class Specification and Visibility** - read, public, private properties.
- **Methods** - how these are defined and managed.
Types

A type generically defines the data structure and the operators that are associated to it. A value is an instance of a certain type.

Types can vary widely in complexity with complex types being constructed using the atomic types, named types, classes and type constructors.

Your class type can be one of the following atomic types:

- **boolean**
  
  The value of a boolean type is either true or false, e.g. 
  
  married: boolean  
  
  can have the value  true or false.

- **char**
  
  The value of the char type is an alphabetical character always given between single quotation marks, e.g.
  
  sex: char  
  
  can have the value  'M' or 'F'.

- **integer**
  
  Integer types have positive or negative integers as their values. e.g.
  
  age: integer  
  
  can have the value  32

- **real**
  
  The value of this type is a real number expressed decimally or exponentially, e.g.
  
  wage_bracket: real  
  
  can have the value  543.35

- **string**
  
  The value of the type string is in fact an alphanumeric character string always given between double quotes, e.g.
  
  last_name: string  
  
  can have the value  "Simpson"

- **bits**
  
  The type bits has a value of a bit string which is a string of bytes of any length that can store binary or bitmap code, e.g.
  
  b: bits  
  
  can have the value  "This is a \0 bits string"
By using the following constructors and by applying them to types you can build up constructed types to define object classes.

You can use the constructors tuple, list, set, and unique set.

- **tuple (...)**

You use a tuple to group together data of different types. You build up tuples using different attributes and types. The basic structure of a tuple is as follows:

```plaintext
tuple (a1: t1, a2: t2, ..., an: tn)
```

where `(a1, a2, ..., an)` is a set of attribute names and `(t1, t2, ..., tn)` is a set of types.

The types used in a tuple can be atomic, constructed or class names, e.g.

```plaintext
tuple (name: string, age: integer, sex: char)
```

could have the following values

```plaintext
tuple (name: "Margot", age: 32, sex: 'F')
```

This is a tuple built up from the atomic attributes `name` of type string, `age` of type integer and `sex` of type char. You can also build up tuples with other tuples, e.g.

```plaintext
tuple (full_name: tuple(first_name:string,
                      last_name:string,
                      title: string),
     age: integer,
     sex: char,
     address: Address)
```

This tuple is constructed from the atomic attributes `age` and `sex` and a nested tuple `full_name` which itself is constructed from three atomic attributes `first_name`, `last_name`, `title`. It also contains a class attribute `Address`.

---

**Note**

Any constructed type, tuple, set or list, can contain any other constructor.
Classes : Types

• list (...)

A list is an ordered collection of objects or values. You build up a list using elements of the same type. The basic structure of a list is as follows:

list (t)

where t is a type.

An example of a list definition:  list (integer)

For example, list (1, 2, 3) is a list with three elements and list () is an empty list.

The order of the elements is important and changes the value of the list, e.g.

list (3, 56, 27, 88) is not the same as list (3, 27, 56, 88)

An element can occur several times and change the list value, e.g.

list (8, 6, 4, 4) is not the same as list (8, 6, 4)

• set (...)

A set is an disordered collection of objects or values. You build up a set using elements of the same type with the following structure:

set (t)

where t is a type.

An example of a set definition:  set (real)

For example, set (1, 2, 3) is a set with three elements and set () is an empty set.

Unlike lists, the order of the elements in a set is not important, e.g.

set (3, 56, 27, 88) is the same as set (88, 27, 56, 3)

However, if an element occurs several times, the set value is altered, e.g.

set (8, 6, 4, 4) is not the same as set (8, 6, 4)

This set is an ordinary set as opposed to the unique set described below.

• unique set (...)

The unique set differs from the ordinary set in that you never have the same element more than once. e.g.

unique set (3, 56, 88, 88) cannot exist, only unique set (3, 56, 88)

O₂ only adds an element to an unique set if it is not already in the set.
class...

You can also use class names as types. These class names can be system-defined in O₂ or O₂ Kit such as Object, Date, Bitmap or they can be any user defined class.

class Address type tuple (street: string,
                           city: City,
                           country: Country)
end

The attribute city is an object of class City.

The class name can also be preceded by the keyword type which means that the specified type has the same structure as the type of the named class e.g.,

class Person type tuple (...
                           address: type(Address))
end

The attribute address is a value of the same type as the object of the class Address.

Example

If you look again at the type part of the example of a class given above:

class Person
type tuple (name:tuple (last_name: string,
                       first_name: string),
           address: type(Address),
           photo: Bitmap,
           age: integer,
           spouse: Person,
           children: list (Person))
end

You see that the class Person is of type tuple with six attributes. The name and address attributes are both nested tuple with the name attribute containing attributes of the atomic type string.

The photo attribute is an object of the class type Bitmap allowing a photo of the person to be included. The age attribute is of the atomic type integer and the spouse attribute is of the class Person. This
Classes : Methods

means it refers to an object of class Person. Finally, the children attribute is a list containing instances of the class Person.

Methods

Program modules that are attached to classes are called methods.

All access to and manipulation of the data contained in objects must be done through the methods - this is the basic principle behind encapsulation of objects.

A method is made up of two parts:

- method signature
- method body

The method signature is in fact the specification of the method and the method body (or code) the implementation of the method.

Note

The method body is implemented separately from the method signature and within the class description you only give the signature of the method.

You code the method in O₂C generally in short single-purpose routines.

• Method Signature

One of the methods specified in our example class Person is:

method public is_adult: boolean

This method is declared inside the class. However you can also declare the method outside the class as:

method public is_adult: boolean in class Person

This is the method signature of the method is_adult and indicates for example whether the person is over 18 years of age.

The method is applied to an object of class Person, called the receiver and returns the boolean value of true or false.
• **Method Body**

The method body corresponding to the method `is_adult`, is defined outside the definition of the class.

```cpp
method body is_adult: boolean in class Person {
    return (self->age>=18);
}
```

The keyword `self` refers to the receiver of the method which is an object of class `Person`. The O₂C keyword `self` is equivalent to the C++ keyword `this`, but differs in that `self` is a mandatory keyword which must be used when the receiver of the method is the same object.

Within other methods after declaring the relevant variable, you can call the method `is_adult`, e.g.

```cpp
o2 Person pers;
b = pers->is_adult;
```

Then ‘pers’ is the receiver of the method. The body defined previously is applied to ‘pers’.

---

**Note**

There are also O₂C functions which are bits of program code that exist in the schema but are not methods and are not included in any application. Refer to the O₂C Reference Manual.
Class Specification and Visibility

A class can have different methods and tuple attributes which are called the properties of the class. There are three kinds of visibilities: private, read and public.

- **private**
  Only those methods attached to the class (and subclasses) in question can read and modify the tuple attribute, or use the method.

- **read**
  Methods from any class can read the value of the tuple attribute but only those methods attached to the class (and subclasses) in question can modify it.

- **public**
  Methods from any class can read the value of the property and can also modify the tuple attribute, or use the method.

**Important**

All class properties are private by default and only become public or read-only when you the developer specifically declare them as such.

To declare a class property as **public** or **read**, simply place the keyword public or read before the property name, e.g

```plaintext
method public is_adult: boolean
```

The entire class can be public, private, or read. The attributes of the tuple can have different visibilities:

```plaintext
class Address public
    type tuple (street: string,
                city: City)
end
```

If you have a class with several different property visibilities, you can also add, where appropriate, the keyword **private** to make the distinction clear.
• **Class Visibility**

The visibility of the class is determined by the private and public parts of the class which represents the internal structure of the class objects and how class operations are implemented, e.g.

```plaintext
class Person
    type tuple( name: tuple(last_name: string,
                              first_name: string),
               photo: Bitmap,
               age: integer,
               read spouse: Person,
               read children: list (Person),
               public dossier_no:real)

    method public is_adult: boolean,
             private add_child(child:Person),
             private salary_bracket:real
end
```

• **Class Specification**

The specification of a class is taken to be the public part of the class including read-only properties. Any methods or attributes dealing with a particular class, i.e. the “clients” of the class, only know therefore the class specification. e.g.

```plaintext
class Person
    type tuple(name: tuple (last_name: string,
                  first_name: string),
               photo: Bitmap,
               age: integer,
               read spouse: Person,
               read children: list (Person),
               public dossier_no:real )

    method  public is_adult: boolean,
            private add_child(child:Person),
            private salary_bracket:real
end
```
Notions of Inheritance and Composition

You can now fully define different classes but different classes can be related to each other in the database structure in two different ways: by composition and by inheritance. These links enable you to define very complex database structures.

- **Class composition**

One class can simply incorporate another class as one of the attributes in its definition. In the example of the class `Person`, you can modify the class by creating another and separate class `Address` for the address and incorporate this class in the class `Person`.

```plaintext
class Person
  type tuple ( name: tuple( last_name: string, first_name: string),
                address: Address, age: integer, spouse: Person, children: list (Person))
end;
/* ----------------------------------------------- */
class Address
  type tuple ( street_address: string, number: integer, city: string, country: string)
end;
/* ----------------------------------------------- */
class Client inherit Person
  type tuple (contact: Person)
end;
```

- **Class inheritance**

A class can also have one or more subclasses and these subclasses are said to inherit the properties of their superclass. For example, as shown above, the class `Client` inherits the properties of the class `Person`. This means that the class `Client` inherits all the five tuple attributes of `Person` and has an additional tuple attribute `contact` of its own. Inheritance is the basic mechanism which allows you to reuse and customize previously specified objects.
2.3 Persistence

Objects and values do not automatically remain in the database after the program that created them has terminated. Programs can create working objects and values which are discarded when the program terminates.

An object or value is persistent if:

- It has a specific name in the database structure.
- It is attached to a named value or a named object in a program.
- It is attached to another persistent object or value in a program.

In other words, any object reachable from a persistent object is persistent. No other objects or values are persistent.

Named Objects

You can name an object using the schema command `name`. You associate a `name` and a `class`:

```plaintext
name Margot: Person
```

The named object `Margot` is created and you can initialize it with the `new` instruction like any other object of the class `Person`. The name gives persistence to the object as well as quick access to it.

Named Values

Set values, as any other value, can be named. You can therefore define a set containing objects of a particular class. Objects which belong to such named value sets are persistent.

For example, a named value, `People`, is defined as a set of objects of the class `Person` and you consequently have access to all the people in the set.

```plaintext
name People: set(Person)
```

and the following $O_2C$ instruction makes the object variable `p` of the class `Person` persistent.

```plaintext
People += set(p);
```

You can also have persistent counters or scalar values, e.g.

```plaintext
name pi: real
```
2.4 Chapter Summary

This chapter has covered the following points:

• **Objects**

The basic element of an object oriented database management system is the object. Each object is made up of a transparent object identity, a value which is in fact the object’s data, and a set of methods that define the behavior related to the data.

• **Classes**

Objects with the same data type and methods are grouped together as a class. A class has a class type, atomic or complex, with different tuple attributes and a set of methods, which form together the properties of the class.

Atomic types can be: boolean, char, integer, real, bits and string. Complex types can be constructed using the tuple, list, set and unique set constructors and applying them recursively to data types or classes.

A class has properties that are either public, read or private. Properties that are public are available to the entire database. Properties that are read can be read by the entire database but only modified by the methods inside the class. And properties that are private can only be read and modified by the methods inside the class.

• **Methods**

Methods are program modules made up of a signature and a body. The signature is the method specification and it is visible outside the class. The body is the method implementation and is specified inside the class.

**Note**

At any time during development you can:

• Add or delete a class.
• Add or delete a class attribute.
• Declare an object persistent.
• Change the visibility of a class property.
• Add, delete a method.
• Modify a method signature.

• **Class Inheritance and Composition**

  There are two ways in which classes can be related to each other.

  A class can incorporate another class as one of its tuple attributes in its definition: this is class composition.

  A class can have one or more subclasses each of which inherits the properties of the class called the superclass.

• **Persistence by reachability**

  To make an object or value persistent, the object or value must be a named object or a named value, or it must be attached to a persistent object or value. This is persistence by reachability.
Now that you have defined your database structure or schema, you can begin programming your O₂ application.

This chapter describes the full implications of class inheritance, and introduces the basics of O₂C and database management in O₂ including persistence and transaction management.

How these data definitions are organized logically and physically in O₂ and what a schema and a base really represent in O₂ are then detailed.

This chapter is divided as follows:

- **Inheritance** - a detailed explanation.
- **O₂C** - what you need to know.
- **Database Management** - Persistence and Transactions.
- **The O₂ System** - definitions of its logical structure and data management.
- **Chapter Summary**.

If you want to start using the system you should refer to the sections on system structure and set up immediately.
3.1 Inheritance

You saw in the previous chapter that different classes can be related to each other in the database schema by composition or inheritance, and that a class can have one or more subclasses and these subclasses are said to inherit the properties of their superclass.

However, inheritance in O₄ is far more varied and enables you to create and manage very complex relationships between classes.

This section extends the notions already outlined and is divided up into the following sections:

- **Class inheritance**
  Extending the notions of simple inheritance and the relationship of the class Object with all other classes in the schema.

- **Method redefinition**
  How can a subclass override or redefine the definition of a method or a tuple attribute that it inherits from its superclass.

- **Late binding**
  The object as an instance of a class determines its own behavior.

- **Static binding**
  When a method of a superclass is applied explicitly to an object.
Class inheritance

An existing class can have one or more subclasses. All objects of the subclass are also objects of the existing class and the subclass type is a subtype of the class type.

The existing class is called a superclass of the new subclass and its type is a supertype of the subclass type.

In the example above, the subclass Employee inherits the characteristics of the class Person.

The class Employee has the same six tuple attributes of Person and its two methods.

However, it also has two additional tuple attributes of its own for position and salary as well as the method change_salary which only applies to Employee.

This relationship between superclasses and subclasses in the schema is called the inheritance hierarchy and is illustrated in Figure 3.13, in which Employee and Client are subclasses of the class Person which in turn is a subclass of Object.
Figure 3.13: Simple class hierarchy

O₂ is initialized with one class called `Object` which is the root of the inheritance hierarchy.

All the classes you define in the schema are directly or indirectly subclasses of the class `Object` and therefore inherit all the system-defined methods attached to this class.

---

**Note**

If a class definition has no `inherit` clause, the new class is automatically a direct subclass of the class `Object`.

---

You can set up inheritance links between two classes that already exist using the command:

```
inherit subclass_name superclass_name
```

You can also destroy inheritance links using:

```
delete inherit subclass_name superclass_name
```

The subclass stops inheriting the properties of the superclass and becomes a direct subclass of the class `Object`.

---

**Redefinition**

If you find that a property inherited from another class does not fit with the use of the subclass, you can simply redefine or override the inherited method or tuple attribute definition. For example, in the class `Employee` below, if you want to define the method `move` differently from the class `Person`, simply define a new definition of the method with the same name in the class `Employee`. 
Inheritance: Late binding

This method definition then overrides the method definition of the same name in the class `Person` for the class `Employee` and any subclasses it may have.

```plaintext
class Person
  type tuple ( name: string,
               address: Address,
               age: integer )

  method move(where:Address),
            add_child (child: Person)
end;

class Full_Address inherit Address
  type tuple (Office_address: Address)
end;

class Employee inherit Person
  type tuple ( position: string,
               address: Full Address,
               salary: real )

  method change_salary,
        move(where:Address)
end;
```

The method signature of the two definitions must be compatible, with the same number of parameters. This means that the parameter type must be identical to or a subtype of the parameter type of the superclass. The subclass method must also return a value whose type is a subtype of that returned by the method of the superclass.

You can also override the definition of a tuple attribute inherited from its superclass either when defining classes or later with an `attribute` command. The same rules apply as for method redefinition.

Late binding

With the redefinition of class methods described above, you now can have a single method name that corresponds to two or more actual methods.
With the above example, a program can send the message `move` to any object of class `Person` and the appropriate code either from `Employee` or `Person` will be executed.

At run time, the system chooses the method depending on the actual class of the receiver.

This concept is known as late binding of methods. In this way it is the object, as an instance of its class, that determines its own behavior.

The system notes the class of the message receiver.

Starting with that class it looks first for an attribute of the specified name and if none exists, for a method of that name.

If again none exists it continues the search in the immediate superclasses and then the next higher level of superclasses until it finds an inherited method of the specified name.

For example,

```
john-> move (new_address);
```

If `John` is an object of class `Person`, its own method `move` is called, but if `John` is an object of class `Employee`, the method `move` of `Employee` is automatically called.

**Static binding**

If you want the method to be that of the class `Person`, and if the object class is a subclass of class `Person`, you must use static binding.

Static binding occurs if a method of a superclass is applied explicitly to an object, even if it has been redefined in the object’s class.

Take the following example:

```
employee -> Person@m;
```

`employee` is an object of the class `Employee` which is a subclass of the class `Person`.

In the example, it is up to you as the programmer to ensure that the method `m` from the superclass `Person` is used and not the method `m` from the class `Employee`. 
Inheritance: Static binding

The most common example is as follows:

```
method body init (name: string,
    age: integer,
    salary: real) in class Employee

{self -> Person@init (name, age);
 self -> salary = salary;}
```
3.2 O2C

This section outlines the basics of O2C with which you can begin to program the procedural commands of O2. Procedural commands are always enclosed inside a body command such as a method body, application body or function body.

The code is compiled and stored when a body command is issued and run when the relevant method, function or application is invoked.

This section covers the following areas:

- Run body
- Objects: new and init
- Values: atomic and complex
- Object value: * operator
- Complex types: -> operator
- Lists
- Sets
- Bits and strings
- Object Sharing
- Copying values
- Generic Display tools

Run body

The O2 command run body allows you to type a quick ad hoc program into the O2 Shell and run and test it immediately.

run body {O2C instructions}

This command is very useful when you are debugging and developing because you can work on an ad hoc program within the framework of a transaction without any restrictions on updating persistent objects.

Note

The run body does not trigger a database commit of preceding schema commands. If you abort, all the schema commands preceding the run body command in the same transaction are abandoned.
Objects: new and init

When you create a new object, you must use the `new` command, e.g.

```cpp
tintin = new Person;
```

This creates a new object of the class `Person`.

The `new` command initializes the object to a default value, by automatically calling a special system supplied method called `init`.

The various types are initialized to the following default values:

- Integers and real numbers to 0.
- Strings and bits to the null string.
- Char to the null character.
- Boolean to false.
- Sets and lists to empty.
- Tuples follow the same rules for each of their attributes.
- Nested tuples are initialized recursively.
- Objects to the nil object.

You can also create an `init` method with or without parameters:

```cpp
method body init(name:string, age:integer) in class Person
{
    self-> family_name=name;
    self-> age=age;
}
```

The `new` operator supplies the parameters which are then sent to the `init` method, e.g.

```cpp
tintin = new Person ("Tintin", 23);
```

As `new` initializes the object, a method can be sent immediately to the object:

```cpp
(new Person ("Captain Haddock", 37)) -> display;
```

**Note**

The `init` method is the only O2 method which cannot be inherited.
Values: atomic and complex

The value encapsulated by an object is associated to a type. Value types can be atomic or complex.

- **atomic**

  Atomic values are: boolean, char, integer, real, string and bits. You declare these values in `o2c` code as

  ```
  o2 integer i, j=4, k;
  o2 boolean a=true, b;
  o2 real x, y=4.8;
  o2 string m, message = "Good morning";
  ```

  In the above example, `j` and `k` are integers with `j` initialized to 4, and `i` and `k` to the default value of 0; the values `a` and `b` are boolean with `a` initialized to `true`, and `b` to the default value of `false`; `x` and `y` are real numbers with `x` initialized to default value of 0 and `y` to 4.8; the values `m` and `message` are strings with `m` initialized to the default `null` string and `message` to "Good morning".

- **complex**

  As was seen in Section 2.2, complex values are constructed with tuple, list and set, e.g.

  ```
  o2 tuple (name: string, age: integer) you;
  ```

  The variable `you` is an instance of the tuple. Tuple attributes are initialized in the same way as atomic values, i.e. integers and real numbers are initialized to 0, characters and strings to null, booleans to false and sets and lists to empty. Nested tuples are initialized recursively and objects are initialized to the `nil` object.

  Tuple attributes are extracted or assigned individually using the dot `.` operator, e.g.

  ```
  you.age = 54;
  ```

**Object value: * operator**

To obtain the value of an object you use the operator `*`. For example, take the following class:

```
class Person
```
public type tuple (name: string,
            age: integer)
end;

and the following code:

```
{ 
  o2 Person tintin = new Person ("Tintin", 23);
  o2 tuple (name:string,
        age:integer) value;
  value = *tintin;
}
```

The * operator separates the value from its object enabling you to reach the object value.

The value of the name attribute (value.name) is "Tintin" and the age attribute (value.age) is 23.

**Complex types: -> operator**

To reach the field of a tuple type object you use the operator ->.

```
object -> attribute;
```

With the class Person above you can use the -> operator in the following way:

```
{ 
  o2 Person tintin = new Person ("Tintin", 23);
  o2 tuple (name:string, age:integer) value;
  value.name = "Tintin";
  value.age=23;
  *tintin = value;
  tintin -> age += 1;
}
```
Consequently \texttt{tintin} -> \texttt{age} is 24 but \texttt{v.age} is 23.

Lists

A list is an ordered collection of objects belonging to the same class, or it is a list of values of the same type. A list can contain the same element more than once. In the following example, the variable \texttt{group} is a list of tuples initialized to the empty list.

\begin{verbatim}
o2 list (tuple (name: string, age: integer)) group;
\end{verbatim}

A list is initialized by default to an empty list: \texttt{list()}. The operations below are the most common on lists. See the Reference manual for all possible operations:

- **Assignment:** \texttt{x = y}
  \texttt{x = list(1, 2)} specifies that the list \texttt{x} contains two elements \texttt{1} and \texttt{2}.

- **Size:** \texttt{count (x)}
  \texttt{count (x)} returns the number of elements in list \texttt{x}.

- **Direct access:** \texttt{x[i]}
  Each list element has an ordinal number with which you directly access the element. \texttt{x[i]} returns the value of the \texttt{i}th +1 element of list \texttt{x}. The first element in \texttt{x} is \texttt{x[0]} and the last \texttt{x[count(x)-1]}.

- **Membership:** \texttt{(p in x)}
  \texttt{(p in x)} gives the ordinal number where \texttt{p} first occurs in list \texttt{x}. If \texttt{p} is not an element of \texttt{x}, a negative value is given.

- **Insertion:** \texttt{x += list(p)}
  \texttt{x += list(p)} adds the element \texttt{p} to the end of list \texttt{x}.

- **Iterations:** \texttt{for (p in x) {instructions}}
  \texttt{for (p in x) \{instructions\}} repeats \texttt{instructions} for each element of \texttt{x}. The variable \texttt{p} is the same type as the elements of \texttt{x}, and it takes the value of each of these elements in turn. You can add a \texttt{where} modifier to limit the number of iterations to a sublist of \texttt{x}:

  \begin{verbatim}
  for (p in x where condition)\{instructions\}
  \end{verbatim}

  The \texttt{instructions} are carried out once for each element of \texttt{x} satisfying the boolean \texttt{condition}.

- **Multiple initialization**
  You can declare a list to have a number of objects as follows:
  \begin{verbatim}
o2 list (Person) dept = list (100: (o2 Person) nil);
  \end{verbatim}
O2C: Sets

The variable `dept` is a list of objects of the class `Person`, initialized to a list of 100 `nil` objects.

Sets

A set is an unordered collection of objects of the same class or a collection of values of the same type. For example,
```
o2 set (integer) lucky_numbers;
```
This defines the variable `lucky_numbers` as a set with the default value of the empty set. You can also declare the set with specific values, e.g.
```
o2 set (integer) lucky_numbers = set (54, 73, 77, 77);
```
In this type of ordinary set, the same object can appear several times. However, you can also declare a unique set in which the same object can only appear once. For example,
```
o2 unique set (Person) kids;
```
The variable `kids` is a unique set of objects of class `Person`, which is initialized to the default empty set `unique set()`. Some of the operations possible on set values where `x`, `y` and `z` are sets are given below. Refer to the Reference manual for all set operations:

- **Assignment**: `x = set (9, 9, 10)`
  
  `x = set (9, 9, 10)` specifies that `x` is an ordinary set containing 3 elements. A set can be converted to a unique set with any duplicate elements being automatically removed, e.g.
  
  ```
o2 set (integer) x = set (5, 7, 5, 6, 5);
o2 unique set (integer) y;
y = (o2 unique set (integer)) x;
y is the unique set (5, 7, 6).
```

- **Size**: `count (x)`
  
  `count (x)` returns the number of elements in set `x`.

- **Membership**: `(p in x)`
  
  `(p in x)` is the boolean `true` if `p` is an element of set `x` and `false` if not.

- **Insertion**: `x += set(p)`
  
  This adds the element `p` to the set `x`. If `x` is a unique set, any duplicate elements are removed and you must write `x += unique set (p)`.

- **Iteration**: `for (p in x) {instructions}`
  
  `for (q in x) {instructions}` repeats the instructions for each element of set `x` and the variable `q`, which is the same type as the elements of `x`, takes on the value of each of these elements in turn. You
can add a `where` modifier if you want to limit the number of iterations to a subset of `x`:

```python
for (q in x where condition) {instructions}
```

The `instructions` are carried out once for each element of `x` satisfying the boolean `condition`.

### Bits and strings

You can declare character strings as follows:

```python
mutable string m, message = "This is a string."
```

This defines the variables `m` and `message` as strings with `m` initialized to the default null string and `message` initialized to "This is a string." These strings end with a zero byte. You can declare bits as follows:

```python
mutable bits q, message = "This is a \0 string."
```

This defines the variables `q` and `message` as bit strings with `q` initialized to the default string of length zero and `message` initialized to "This is a \0 string."

#### Note

The difference between a character string and a bit string is that any byte of a bit string can contain any value including zero and can therefore include one or more embedded zeros.

You can use the following operations on both string and bits variables. For all the operations available, please refer to the Reference manual:

- **Assignment**: `x1 = x2`

  `x1 = x2` copies the string or bits variable `x2` into `x1`. However the following operations are different:

  ```python
  mutable string s;
  s = "Assigned string";
  ```

  places the string "Assigned string" that ends with a zero byte into the string `s`.

  ```python
  mutable bits b;
  strncpy(b, "string with \0 in it", 19);
  ```

  places the bit string: "string with a \0 in it" into the bit string `b`.

- **String length**: `count(s), count(b)`
count(s) returns the length of string s and does not include the zero byte that ends the string.

count(b) returns the exact length of the bit string b including any embedded zero bytes.

- **Comparison:** x1 == x2, x1!= x2
  
x1 == x2 is true if the string or bits x1 is equal to x2.
  
x1 ! = x2 is true if the string or bits x1 is not equal to x2.

- **Concatenation:** X1 + X2, X1 += X2
  
  X = X1 + X2 causes X to be equal to the concatenation of the string or bits x2 onto the end of x1.
  
  X1 += x2 or X1 = X1 + x2 appends x2 onto the end of x1.

- **Iteration:** for (C in x) {instructions}
  
  for (C in x) {instructions} repeats the instructions for each character in the string s or bits x and the char variable c takes on the value of each character of x in turn.

  You can add a where modifier to limit the number of iterations:
  
  for (C in x where condition) {instructions}

  The instructions are carried out each time the condition is met.

---

**Note**

Variables of type string and bits behave differently from C strings of type char *.

However, a string variable or a bits variable may be passed to any C library function which calls for a parameter defined as const char *. A library function cannot alter the contents of the string or bits. See O2C Reference manual for more details.
Object Sharing

An object can be shared but a value cannot.

```cpp
class Person
type tuple (name: string,
    age: integer,
    mother: Person,
    father: Person,
    spouse: Person,
    children: list(Person))
end;
```

An object of the class `Person` contains several other objects of class `Person` as some of its tuple attributes. Any one of these objects can be a component of several other objects.

For example, an object representing a person John appears as the `spouse` attribute of the object representing his wife and as a `father` attribute for his children and as an element of the `children` list of his parents.

All these references are to the same physical object John. If the object John is modified, e.g. his age, in the object representing his father then the object John is automatically updated in all the other objects.

Copying values

You cannot share values but you can copy them.

For example, if objects `p1` and `p2` of class `Person` are set by `p1->name = p2-> name = "Joe"`, the two objects have the same value. But if later `p2->name = "Dupont"`, the two objects now have different values.

**Important**

Objects have identity and can be shared, values have no identity and cannot be shared.
Generic Display tools

O2 has a number of predefined display functions and methods. These tools are:

- Display method.
- Edit method.
- Display function.
- Input function.

The functions apply to values and the methods apply to objects. The examples given below are all based on the following class Person.

```eu
class Person
public type tuple ( name: string,
                   age: integer,
                   position: string,
                   spouse: Person,
                   children: list (Person) )
end;
```

If you are using O2Look, these methods and functions allow you to display the following types of window.

In Figure 3.14, the object Tintin is displayed on the left and the string value Tintin is displayed on the right. The same principle applies to all O2Look windows. If you can carry out modifications in the window there is always a pencil and an eraser button. If you can only read the contents of the window, there is only an eraser button.

To save your modifications, click the pencil button. The value is updated and the presentation disappears. If you click on the eraser button all modifications are ignored and the presentation disappears.
• **display method**

The *display* method displays the public and/or read-only value of the receiver object which you cannot modify:

```
receiver -> display; or receiver -> display();
```

For example:

```
run body { 
tuple (name: Tintin,
        age: 24,
        position: "Rave_reporter",
        spouse: [Person],
        children: list ([Person],
                      [Person]))
 }
```

If you are using O₂Look, you see the window shown in Figure 3.15. To exit, click on the eraser button:

![Figure 3.15: display method](image)

If you are using an alphanumeric environment you see:

```
tuple (name: Tintin,
        age: 24,
        position: "Rave_reporter",
        spouse: [Person],
        children: list ([Person],
                      [Person]))
```

• **edit method**
The edit method is only available with O2Look and is the same as the display method, except that you can modify the contents and only public properties can be updated.

For example:

```java
run body { o2 Person p = new Person, q = new Person, 
          r = new Person, s = new Person;
          p -> name = "Tintin";
p -> age = 24;
p -> position = "Rave_reporter";
p -> spouse = q;
p -> children = list (r, s);
p -> edit; }
```

This gives the window displayed in Figure 3.16.

![Figure 3.16: Edit method](image)

If you click on the methods button you see all the public methods of the class Person. You can then interactively use the methods.
• **display function**

The display function displays a value generically.

You cannot modify the display.

\[
\text{display (expression)}
\]

For example:

```java
run body {
    o2 Person p = new Person;
    p -> name = "Tintin";
    display (p->name);
}
```

In **O₂Look** you now see the following window appear:

![Figure 3.17: display function](image)

*Figure 3.17: display function*
• *input function*

The input function displays a template presentation based upon the type definition of the value to be entered.

```c
input (variable);
```

For example:

```c
runcycle {
  o2 Person p = new Person;
  input (p->name);
  p->display;
}
```

In O2Look, this displays the window shown in Figure 3.18 displaying the initialized default value.

![Figure 3.18: input function](image)

Enter a name and click on the pencil button. The value is now updated and the presentation disappears. If you click on the eraser button all modifications are ignored and the presentation disappears.

If you type in the name Tintin and click on the pencil button, you see the following window:

![Figure 3.19: Display of the new person](image)
3.3 Database Management

As a database management system O₂ has all the usual features associated with database management:

- Disk management for storing objects and values transparently.
- Concurrency control to avoid user conflicts.
- Ensure consistency in a multi-user environment.
- Crash recovery to ensure that an uncorrupted database survives hardware and software failures.
- Interactive query language.
- System tuning facilities.

Objects and values are stored in the database using two mechanisms: persistence by reachability and transaction management.

Persistence

The notion of object persistence was introduced in Section 2.3. Objects can be persistent or transient in the system. An object is persistent if it remains in the database after the transaction in which it was created is successfully terminated.

Every named object and every named value in the database is persistent and every component of a persistent object or persistent value is also persistent.

The most common way to make an object persistent is:

- To give the object a specific name.
- To make the object a component of another persistent object.

If you want to delete a persistent object or value from the database you must cut it off from all its roots of persistence. This means removing all its name associations. To remove the name of a persistent object or value you use the delete name command. If other persistent objects or values point to the persistent object you must break these links, and if any copies of persistent values exist elsewhere these values continue to exist but without a name of persistence.

Schema definitions are always named and are therefore persistent. Class definitions, named type definitions, methods, applications, application programs, application variable definitions, functions, volumes, schemas, bases, index names, cluster names and version names are all persistent.
Transactions

An O₂ session can contain several transactions.

There are two run modes in O₂:

- **transaction mode**

  The transaction mode is a run mode in which all types of access to the database are allowed (read, write and update), with concurrency control ensuring that two clients work properly together at the same time.

  A transaction is defined as an atomic unit of database and/or schema modifications which are written to the database as a whole or abandoned as a whole.

  If anything goes wrong during a transaction whether it be a hardware failure, operator error, database deadlock or a programming mistake, all the changes made to the database during the transaction are reversed and consequently data integrity is preserved at all times.

  You can only have one transaction per user at any one time.

- **read-only mode**

  The read-only mode is a run mode with no concurrency control in which you have read access to the database.

  You use this mode if you want to read the database without being affected by the update of other transactions.

  The objects of the base can be read but not modified. There is no locking, and locks owned by other users are not checked.

  An O₂ session launched by the administrator or development tools starts by default in transaction mode. On the other hand, an application starts in read-only mode.

  During a transaction you can carry out the following operations no matter which interface you are using to O₂ (O₂Engine API, command language, or any of the programming languages O₂C, C, C++, Java):

  - **commit**

    This ends the current transaction and commits any modifications made. It frees the memory taken up by O₂ references or handles to objects and releases all locks. In the command language, it immediately starts a new transaction.

  - **validate**
This ends the current transaction, commits any modifications made and releases all locks but it does not free the memory taken up by $O_2$ references or handles to objects. In the command language it then starts a new transaction.

- **abort**

  This cancels all the updates carried out during the current transaction. It frees the memory taken up by $O_2$ references or handles to objects and releases all locks. If used in the command language, it immediately starts a new transaction.

- **transaction**

  This is used in read-only mode. It starts a transaction and cleans the persistent object buffer thereby ensuring that persistent objects have the most up-to-date value.
3.4 The O₂ System

You can look at the structure of the O₂ system logically and physically.

As shown in Figure 3.20, in O₂, the data definitions you saw in Chapter 2 are grouped together into one or more logical entities called schemas, each of which has one or more associated bases which contain the actual objects and values.

Physically, these schemas and bases are recorded on files grouped together into volumes, which are themselves grouped into named systems. A volume is implemented as a Unix file.

This section now details the logical structure of the system by defining schemas and bases, and outlining how to import and export schemas. It then describes how the database is physically managed including definitions of the O₂ named system, volumes, catalogue, log and shadow. For further details see the O₂ System Administration Guide.

Schema

The various different definitions of the O₂ data model are grouped together into a schema.
The schema describes the structure and behavior of the associated $O_2$ bases and is made up of the following elements:

- Class definitions including specifications of class types and methods including method code as well as the different types of relationships between these classes.
- Named objects and named values definitions.
- Application definition including program code and the variables these programs use.
- Function definitions including function code.

You can create any number of logically separate schemas.

You can create and name a schema explicitly using the `create schema` command and delete it using the `delete schema` command.

You can modify the schema when you are developing using $O_2$.

For example, a schema $S$ contains all the classes, methods and functions for an application managing a set of Person.

![Figure 3.21: Logical structure: Schemas and bases](image-url)

**Base**

A schema has one or more bases associated to it as shown above in Figure 3.21.

A base contains data, i.e. the objects and values whose structure and behavior conform to the schema descriptions that is the class and method definitions in the schema.

To declare and name a base, you use the `create base` command, and to destroy a base and all the data related to it, use the `delete base` command.
The O2 System: Importing a schema

You can have any number of logically separate bases but remember that a base is associated to one schema only.

Moreover, you work on one schema and one base at any one time. You can define and modify a schema without referring to a base but when using a base it is always associated to a schema.

For example, you can attach one base to the schema S in order to manage your customers and one base to manage your suppliers.

Importing a schema

You can use all or part of another schema to build up your schema by importing one or more of its classes with the `import schema` command.

This command allows you to share class definitions and named object and named value definitions between different schemas.

Note that only the schema definitions are imported not the actual data.

For example, to make the predefined classes `Date` and `Bitmap` of O2Kit available to the working schema you use the following command:

```
import schema o2kit class Date, Bitmap
```

**Note**

You cannot modify an imported class but you can add subclasses to it and modify the methods and type structures in these subclasses.
Like a class, a schema has public and private components. The public component (the exported classes) can be used in other schemas. The implementation of the schema (non-exported classes and implementation of all methods) is hidden for other schemas and cannot be modified by them.

To be imported, a class must have been exported by the schema which owns it.

**Named System**

In $O_2$, schema definitions and data are physically managed in a named system.

![Diagram of Logical and Physical Structure of a Named System]

*Figure 3.23: Physical structure*

A named system is a collection of volumes that is treated as a single unit by $O_2$ and into which such volumes can be added, removed or reorganized. You find the following types of volumes in a named system:

- **User volumes**
  These are the physical Unix files on which $O_2$ data and schema information is recorded.

- **Catalog volume**
The O2 System: Named System

This holds the size and location of all the physical volumes and lists the schema-base associations.

- **Shadow volume**
  The shadow provides temporary disk space in which to store O₂ objects.

- **Log volume**
  The log enables recovery after any sort of system failure.

For further details see the O₂ System Administration Guide.
3.5 Chapter Summary

This chapter has covered the following points:

• **Inheritance**

  In O₂, you can create simple to very complex relationships between classes using simple inheritance, method redefinition, late and static binding, multiple inheritance, renaming and repeated inheritance.

• **O₂C**

  To start programming in O₂C, you must know how to use a run body, the init and new commands, and the ".", "=" and "*" operators. The value encapsulated by an object is associated to a type. Value types can be atomic or complex. Such a value can be copied but it cannot be shared. An object in O₂C can be shared however. There are predefined methods including: display and edit, and predefined functions including display and input.

• **Database Management**

  Database management includes persistence of database objects and transaction management. An object is persistent if it remains in the database after the transaction in which it was created is successfully terminated. A transaction refers to a set of changes to a database; these changes are processed together in their entirety. Transactions in O₂ include: validate, abort, commit, quit and transaction.

• **The O₂ System**

  The logical system structure of O₂ is based on data definitions being grouped together into one or more logical entities called schemas, each of which has one more associated bases which contain the actual objects and values. Physically these schemas and bases are recorded into volumes (Unix files), which are themselves grouped into named systems.
This chapter presents a step-by-step tutorial, showing how to start the O₂ system, how to create and populate a simple database, how to write methods, how to link them into applications, and how to run queries. We do not attempt to explain everything that goes on here, nor do we present an exhaustive description of programmers’ options.

It is divided into the following sections:

- Creating an O₂ system
- Launching O₂
- Creating and populating the database
- Beginning to program your database
- Methods
- Starting to query your database
- Linking methods to applications
4.1 Creating an O₂ system

As described in Section 3.4, schema definitions and data are physically managed in a named system as shown in Figure 4.24.

Each named system groups together various user, catalog, shadow, and log volumes.

Before you can start an O₂ session in your new named system, you or your system administrator must first set up the system.

For example, take the following parameters:

server: machine_15
system: demo

The first thing you must do is create one or more directories to store in the files of the system demo:

mkdir /disk1/o2vol_1
mkdir /disk2/o2vol_2

You must now create entries in the .o2serverrc file which is in the O₂ installation directory.

For further details see the O₂ System Administration Guide.
Creating an O2 system

This file is an ASCII file in the O2 installation directory and contains the description of each named O2 system in use. The description has the following structure:

```
[system_name.]
server = server_name
[cataldir = catal_path]
[logdir = log_path]
[shadowdir = shadow_path]
```

In the example, the entries are created in the .o2serverrc file for the new system demo:

```
demo.server = machine_15
demo.cataldir = /disk1/o2vol_1
demo.logdir = /disk2/o2vol_2
```

There is a bin directory in the O2 installation directory. This bin directory contains the following program files: o2shell, o2dba_init and o2server.

To initialize the system, use:

```
o2dba_init -system system_name
```

The o2dba_init command destroys any data in the database system specified by system_name if such a system already exists.

This is useful when you want to restart with no data.

There are numerous system classes and methods you can utilize which must first be loaded using:

```
o2dba_schema_load -name schema_name -directory $O2HOME/o2schemas -system system_name -sources
```

**Note**

To perform a schema load after starting an O2 server, you must relaunch the server in single user mode.

```
o2server -system system_name -mode single-user
```
4.2 Launching $O_2$

To connect the server to the system and start the $O_2$ server:

```bash
$O2server -system system_name
```

Do this in another window so that the server messages are not mixed up with other messages.

To start up a session once the server is running and bring up the $O_2$ shell:

```bash
$O2shell -system system_name
```

An $O_2$ session always operates on a single named system. You now see the standard prompt

**Type your command and end with ^D**

This gives you $O_2$ in alphanumeric mode.

You can also add the verbose option which repeats each of your commands. To use this option, when you start up the session, type:

```bash
$O2shell -system system_name -verbose
```

**Note**

This tutorial is given in alphanumeric mode and does not use the verbose option.

However, if you have $O_2$ Tools as part of your configuration you should start up your session as follows:

```bash
$O2tools -system system_name
```
4.3 Creating and populating the database

Now you have set up and launched O₂, you can now begin to define your database schema and populate your database.

Create schema

Before doing anything you must create or set a schema which contains your schema definitions.

To create a schema s₁, type:

```
create schema s1
^D
```

Always end this and all other O₂ commands with Return followed by control-D. You now see:

```
Type your command and end with ^D
```

The schema s₁ is now the current working schema. To display what this schema contains, type the command:

```
display classes
^D
```

O₂ displays what is contained in the schema s₁:

```
Object
```

The schema s₁ is not empty but contains a class that is automatically supplied by the system: Object. This class is at the top of the class hierarchy and all other classes are inherited from this object.

---

**Note**

All other classes in the schema are subclasses of the superclass Object.
Set schema

If you want to use a schema that already exists, simply type:

```
set schema s1
^D
```

Create class

Now create the class `Person`.

All class names begin with a capital letter as O₂ names are case-sensitive. Type the following few lines, ending as always with a carriage return and control-D.

```
class Person
type tuple (name: tuple (last_name:string,
first_name:string),
birthdate: Date,
spouse: Person,
children: list (Person))
end
^D
```

Although the format is not critical we strongly recommend you space your classes as before for a more legible code.

The class `Person` has been created.

To see the class hierarchy, type

```
display classes
```

O₂ displays:

```
Object
   Person
```

This indicates that `Person` is a subclass of `Object`, and consequently all objects of the class `Person` are also objects of the class `Object`.
Creating and populating the database

If you type the following command:

display class Person

you see the following:

```plaintext
class Person /* partially defined */ inherit Object
private type tuple (name:tuple (last_name: string,
    first_name: string),
    birthdate:Date, /* undefined */
    spouse: Person,
    children: list (Person))
end;
```

This tells you that the class `Date` is unknown. Not surprising really as you have not created it yet! However, you can import it from O2Kit using the command (recall that you must first do an `o2dba_schema_load`):

```plaintext
import schema o2kit class Date
```

If you now display the class `Person` you see the following:

```plaintext
class Person  inherit Object private type
    tuple (name: tuple (last_name: string,
        first_name: string),
        birthdate:Date,
        spouse: Person,
        children: list (Person))
end;
```

Note that the partially defined comment and the undefined comment have disappeared.

With O2 you can experiment with a partially-defined schema; everything acts normally, as long as you do not try to execute a method that uses an undefined part of the schema.

If you do, you will get an error message about a `nil` object.
Using a text editor such as emacs, type the definition of the class Address:

```
class Address
  public type tuple (street_address: string,
                      city: string,
                      country: string,
                      postal_code: string)
end;
```

You can now copy and paste from emacs to O2 and create the class as with class Person. Or, you can use the following command

```
"path/file_name"
```

where `file_name` is the name of the file containing the above class and `path` is the full path of the subdirectory.

To add this class as an attribute to `Person` you can simply redefine class `Person` by reentering all the information plus the new attribute.

This can be complicated and long for very big classes with lots of attributes. However you can also use the command:

```
create attribute address: Address in class Person
```

O2 creates the attribute.

The command `display class Person` confirms that the tuple structure of that class now has a fifth attribute.

```
class Person inherit Object private type
tuple(name: tuple(last_name: string,
                   first_name: string),
       birthdate: Date,
       spouse: Person,
       children: list(Person),
       address: Address)
end;
```
Creating and populating the database

Named objects and values

If you now display all the classes of your schema s1 you see the following hierarchy:

Object
  Person
  Date /* imported from o2kit */
  Address

From the superclass Object, you now have three classes. However, you still have no objects in your database.

Remember a class is the description of a group of objects with the same characteristics - it is not a repository for them. To store and make your objects persist in the database, you need to create a repository.

In our example for objects of the class Person, you can create a repository or persistent name called People. Type:

    name People: set (Person)

O2 creates the name.

You can display the name using display name People and you see that People is a repository containing a set of objects of the class Person:

    name People :set(Person);

People is neither a class nor an object but a named value that is associated with a group of objects of the class Person. Note that not all the objects of class Person necessarily belong to the set People. The set(Person) is initialized to an empty set by default ready to hold the objects you will create later on in this tutorial.

With such a repository or persistent name you can make objects persist by simply associating them to it. The objects are thereby stored permanently on disk. The simplest way to associate an object to a name is to include it in a named set.

Note that all objects of the other two classes Address and Date are likely to be attributes of objects of class Person, and consequently benefit from the persistence of the objects of the class Person because objects which are components of persistent objects are also persistent.
Confirm the schema modification.

You now need to confirm the schema definitions in order to create a base that will contain the data of this application. Type:

```
confirm classes
```

Create base

You have just created a database schema `s1` under which you can now define your schema classes.

```
create base b1
```

This specifies a new database called `b1` whose structure and behavior is governed by the current schema `s1`.

You now have a schema and a base which now apply for all the commands you now make.

Set base

If you want to use a base that already exists, simply type:

```
set base b1
```

Figure 4.25: Schema and bases
4.4 Beginning to program your database

To summarize the first part of this chapter:

You created a schema to hold your schema definitions:

```o2kit
create schema s1
^D
```

You created the class `Person`, imported the class `Date` from o2kit and created another class `Address`.

```o2kit
class Person inherit Object public type
tuple(name: tuple(last_name: string,
   first_name: string),
   birthdate: Date,
   spouse: Person,
   children: list(Person),
   address: Address)
end;
^D
```

You created persistent name `People` in which to hold objects of the class `Person`.

```o2kit
name People: set(Person)
^D
```

Note that the class Person is a private class. To change it to a public class, type:

```o2kit
public type in class Person
^D
```

Now external programs can access the attribute values and methods of the class `Person`. The class `Address` is already public.

You created a base in which to store objects:

```o2kit
create base b1
^D
```
Add an object to the database

To add a particular object to the database, you can use the following type of program:

```c
run body{
  o2 Person p = new Person;
  p->name.last_name ="Jones";
  p->name.first_name ="Indiana";
  p->birthdate = new Date (30, 03, 15);
  p->address = new Address;
  p->address->street_address ="Elm Avenue";
  p->address->city = "Little Rock";
  p->address->country = "USA";
  People += set(p);
  p->edit;
}
```

This program declares the variable p of type Person and creates a new object of class Person. The `new` command creates a new object and initializes it to the default value.

`set(p)` is then added to the set People in order to make the object p persistent. You must use the construct `set(p)` because you can only add a set to a set.

You specify some of the attributes of p to specific values. Two of the attributes are also objects and you must create them using the `new` command.

You now see on the screen the window shown in Figure 4.26 below.
Component objects are displayed as icons, and by default this icon contains the class name of the object. You can redefine this default by redefining the `title` method. This is described in a section below.

If you click on the Date or Address using the right mouse button you get a pull down menu in which you can display the object. Date is set to the date and address to the address both specified in the run body.

```
run body{
    o2 Person p = new Person;
    p->edit;
    People += set(p);
}
```
This program declares the variable \( p \) of type \texttt{Person} and creates a new object of class \texttt{Person}. The \texttt{new} command creates a new object and initializes it to the default value.

\texttt{set \( p \)} is added to the set \texttt{People} in order to make the object persistent. The following window appears:

![Presentation of the object Person](image1)

Figure 4.28: Presentation of the object Person

You can now fill out this window. Each black box signifies an empty object. To create a new object click on the box using the right mouse button and \texttt{new} appears. Release the mouse button when on \texttt{new} and the specified object window appears. After editing and saving this widow, the black box is replaced by the object type, e.g. \texttt{Address} or \texttt{Date} or \texttt{Person} as in Figure 4.29.

![Partially completed Person object](image2)

Figure 4.29: Partially completed Person object
Now when you click on this area using the right mouse button you see a pull down menu which is the same as the methods menu.

These are predefined methods from the superclass `object`. The method `edit` automatically creates an X window presentation for the modification of the object.

When you click on the pencil button, the modifications are recorded and the `edit` method returns the integer value defined as `LK_SAVE` to the program.

In the spouse area, click on `Person` and select `edit` in the menu that appears. The same window as in Figure 4.28 appears. The same applies for the children objects. Fill out these windows and click on the pencil button.

Using the right mouse button click on the Date area and select `edit`. The window shown in Figure 4.31 appears.
When you edit the Address object, you see the following window:

![Figure 4.32: Address object](image)

To save all your modifications click on the Pencil button of each window. If you want to see what is in the set People, type:

```java
run body {
    display (People);
}
```

You now see:

![Figure 4.33: The set People](image)

This shows that you have added four objects of class Person to the database. To see the objects click on the Person area using the right mouse button and choose edit or display.
4.5 Methods

To tell the system how these objects behave, you write and add various methods.

If you type:

\texttt{display methods in class Person from Object}

You see the list of all the predefined methods available to you including \texttt{refresh_all, erase_all, display} and \texttt{edit} that are automatically included in the methods menu in the presentation of the object of the class Person.

This section outlines the following methods:

- \texttt{add_child}
- \texttt{marriage}
- \texttt{Init}
- \texttt{title}

\textbf{add_child}

This method enables you to add a child to your receiver object.

It takes the last name and address of the receiver object, i.e. the parent, and adds the new child to the list of children.

Type the method signature:

\begin{verbatim}
method public add_child (first_name:string):Person in class Person
\end{verbatim}

\texttt{(first_name:string):Person} indicates that the method \texttt{add_child} requires the first name of the child and returns a \texttt{Person}.
Now type the method body:

```java
method body add_child (first_name:string) : Person in
class Person{
    o2 Person c = new Person;
    c->name.last_name = self->name.last_name;
    c->name.first_name = first_name;
    c->address = self->address;
    self->children += list(c);
    c->edit;
    self->refresh_all;
    if (self->spouse != nil) {
        self->spouse->children += list(c);
        self->spouse->refresh_all;
    }
    return c;
}
```

This method is now added to the methods menu for each presentation of the object Person. When you add a child to your parent object using this method, the following window appears:

![Figure 4.34: first_name of child](image)

Enter the first name and click on the pencil button. The presentation of the new Person is now displayed with the `last_name` attribute and the address already completed as in Figure 4.35 below.

![Figure 4.35: add_child method](image)
marriage

Much like the method `add_child`, the method `marriage` enables you to add a spouse to your receiver object. This method takes the address of the receiver object, i.e. the husband or wife.

It also takes the receiver object itself as the spouse of the new object. It then adds the spouse to the repository `People`.

Type the method signature:

```plaintext
method public marriage (first_name:string, last_name:string):Person in class Person
```

This method requires the first name and last name of the spouse and returns a `Person`. Now type the method body:

```plaintext
method body marriage (first_name:string, last_name:string):Person in class Person {
    o2 Person w = new Person;
    w->name.last_name = last_name;
    w->name.first_name = first_name;
    w->address = self->address;
    self->spouse = w;
    w->spouse = self;
    People+= set (w);
    self->refresh_all;
    return w;
}
```

This method is now added to the methods menu for each presentation of the object `Person`.

When you launch the `marriage` method to your receiver object, the following window appears:

![Figure 4.36: First name and last name](image)

*Figure 4.36: First name and last name*
init

If you want to set up the object of the class `Person` when it is created, you can create the `init` method.

**Note**

The method `init` is automatically called by the constructor `new`, if it is defined for the class.

For example if you want all objects of the class `Person` to be placed in the repository `People` and to set the birthdate to today’s Date.

Type the method signature:

```
method private init in class Person
```

Type the method body:

```
method body init in class Person
{
    self->birthdate= new Date (0, 0, 0);
    People+= set (self);
}
```

Now when you run:

```
run body{
    o2 Person p = new Person;
    p->edit;
}
```
you get the following presentation in which the birthdate is not an empty object:

![Figure 4.37: Presentation after redefinition of init method](image)

**title**

When an object is displayed using O₂Look, the presentation title is the result of the title method. This method is defined in the Object class and is inherited by all classes. It returns the class name of the object.

---

**Note**

This is a method that is called up automatically by O₂Look every time you display an object. It must be called `title` and it must return a `string`.

You can redefine the `title` method to give a more meaningful string for your application.

For the class `Person`, type the method signature:

```plaintext
method public title: string in class Person
```
and the following method body:

```cpp
method body title: string in class Person
{
    return self->name.first_name + " " + self->
        name.last_name;
}
```

This displays the first and last name of the object of class Person.

You can also define this method for the class Address. Type the signature:

```cpp
method public title: string in class Address
```

and the method body:

```cpp
method body title: string in class Address
{
    return self->city + " " + self->country;
}
```

This displays the city and country instead of the word Address.

If you now display an object of class Person using the `edit` method, the result is seen in the presentation of a Person object, as in Figure 4.38:

![Figure 4.38: Presentation after the redefinition of the title method](image-url)
Methods

You can also see how useful this method is if you rerun the following program:

```
run body {
    display (People);
}
```

You now see the names of all the people you have added to the database thereby facilitating access to them.

*Figure 4.39: People*
4.6 Starting to query your database

You can start using OQL in the alphanumeric environment or in the O₂C shell when you see the following message:

Type your command and end with ^D

To run OQL, type:

query
^D

You must type ^D (Control - D) on a separate line. You now see:

Query Interpreter
Type your query and end with ^D

Type your query, ending it with ^D. For example:

People
^D

This query simply calls the entry point People. An entry point is a named object or named value. You see the following list of people:

![People](image)

Figure 4.40: People

To type another query you must first erase the previous one.

A query you will use a lot is the "select from where" clause which enables you to extract those elements meeting a specific condition from a list or set.

The OQL query has the following structure:

- **select**: defines the structure of the query result.
- **from**: introduces the sets or lists against which the query runs.
- **where**: introduces a predicate that filters the result.
For example,

```sql
select p->name.first_name
from People p
where p->name.last_name = "Jones"
```

To leave the query session type:

`^D` (or `quit`)
4.7 Linking methods to applications

You must organize your program into an application. Create the application `Company_Personnel` and add a program called `add_employee`.

```o2
application Company_Personnel
program public add_employee
end
^D
```

You must create the repository `Staff`:

name Staff: set(Person)

Now write the program body:

```o2
program body add_employee in application Company_Personnel{
    o2 Person p = new Person;
    o2 integer x;
    x= p->edit;
    if (x== LK_SAVE)
    {
        transaction;
        Staff+=set(p);
        validate;
    }
    p->display;
}
^D
```

Run the program:

```
run program add_employee in application Company_Personnel
```

This program creates a new object of the class `Person` and displays it using the edit method.

When you click on the Pencil the new object is added to the repository `Staff`. The object is then displayed again.
Run application

Now if you want to see how your application works, type

run application Company_Personnel

You see the window:

![Application Window](image)

*Figure 4.41: Application*

If you click using the right mouse button, you see a pull-down menu displaying the program add_employee. Release the mouse button.

The following window appears:

![Add Employee Window](image)

*Figure 4.42: Add an employee*

If you play with the application, you will find that the application is far from perfect!

For example, in order to work you must fill out the initial person's details and save them in order to use the add child and add spouse methods in the correct way.

Try to sort this problem out!
Adding programs

To access the objects in your database, you can add various programs.

For example, to display the employees is very easy:

Write and run the program:

```
program public display_employees in application
  Company_Personnel;
```

with the body:

```
program body display_employees in application
  Company_Personnel  { display (Staff);
  }
```

This program is now included in the list of programs of the application shown in Figure 4.43.

This ends the tutorial and the O2C Beginner’s guide. You should now know the basics to begin programming and developing different applications using O2C and the O2 object-oriented database system.

For a complete guide to this powerful 4th generation language, please refer to the O2C Reference manual as well as the other documentation for all the O2 modules. Remember, however, for a full understanding of O2 and O2C, you must follow our specially designed training courses.

Other services to help you design, build and develop your applications are also available with O2 Technology, including a Hotline and different Consulting Services.
INDEX
INDEX

Symbols

.o2serverrc 70

A

Abort transaction 62
Application 94
  Run 95
Architecture
  Client/server 14
  O2 12
Atomic
  Type 27
  Value 48
Attributes 28, 80

B

Base 64
  Create 78
  Set 78
Bits 27
  bits 52
Boolean 27

C

C 13
C++
  Interface 13

Catalog volume 66
Char 27
Class 26
  Composition 35
  Create 74
  Definition 25
  Display 75
  Inheritance 35, 73
  Methods 26
  Object 73
  Properties 33
  Specification 33, 34
  Type 26
  Visibility 33, 34
Class inheritance 40, 41
Client/server architecture 14
Commit transaction 61
Complex value 48
Composition 35, 40
classify classes 78
Constructors 28
Create
  Attribute 76
  Base 78
  Class 74
  Schema 73

D

Data encapsulation 10
Database
  Management 60
  Object 80
  Query 92
Display
  Class 75
  Function 58
  Method 55, 56, 83
# INDEX

<table>
<thead>
<tr>
<th>E</th>
<th>L</th>
</tr>
</thead>
<tbody>
<tr>
<td>Edit method 55, 56, 83</td>
<td>Late binding 40, 43</td>
</tr>
<tr>
<td>Encapsulation 10, 31</td>
<td>List 29, 50</td>
</tr>
<tr>
<td></td>
<td>Log volume 67</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>F</th>
<th>M</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function</td>
<td>Method 24, 31, 85</td>
</tr>
<tr>
<td>Display</td>
<td>Body 32, 86</td>
</tr>
<tr>
<td></td>
<td>Display 55, 83</td>
</tr>
<tr>
<td></td>
<td>Edit 55, 83</td>
</tr>
<tr>
<td></td>
<td>Init 88</td>
</tr>
<tr>
<td></td>
<td>Redefinition 40, 42</td>
</tr>
<tr>
<td></td>
<td>Signature 31, 85</td>
</tr>
<tr>
<td></td>
<td>System supplied 83</td>
</tr>
<tr>
<td></td>
<td>Title 89</td>
</tr>
<tr>
<td></td>
<td>Methods Menu 83</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>I</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td>Import 75</td>
<td>Named</td>
</tr>
<tr>
<td>Inheritance 35, 40</td>
<td>Objects 77</td>
</tr>
<tr>
<td>Init Method 47, 88</td>
<td>System 66</td>
</tr>
<tr>
<td>Input Function 59</td>
<td>Values 77</td>
</tr>
<tr>
<td>Installation directory 71</td>
<td>New command 47</td>
</tr>
<tr>
<td>Integer 27</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>J</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java 13</td>
</tr>
</tbody>
</table>
Object 10, 73
  Database 80
  Encapsulation 31
  Identity 24
  Integration 11
  Persistence 36
  Sharing 54
  Structure 24
  Value 48
Operator * 48
Operator -> 49
OQL 13, 17
  select..from..where 93
  Tutorial 92
Persistence 60
Physical structure 70
Private
  Class 79
  Property 33
Program 80
  Body 94
  Run 94
Programming languages 16
Property
  Private 33
  Public 33
  Read 33
Public
  Property 33
  Type 79
Query language 92
INDEX

R

Read-only 33
Real 27
Redefinition 42
Run
  Application 95
  Program 94
Run body command 46, 80

S

Schema 63
  Create 73
  Import 65, 75
  Set 74
Select from where 92
Session 70
Set 29, 51, 77
  Base 78
  Schema 74
Shadow volume 67
Static binding 40, 44
String 27
string 52
Superclass 40
System
  Architecture 12
  Initialization 71
  Schema load 71
  Set up 70
Transaction 61
  Abort 62
  Commit 61
  Validate 61
Tuples 28
Type 27
  Atomic 27
  Bits 27
  Boolean 27
  Char 27
  Class name 30
  Complex 49
  Constructed 28
  Integer 27
  List 29
  Real 27
  Set 29
  String 27
  Tuple 28
  Unique set 29

U

Unique set 29
User volume 66

V

Validate transaction 61
Value 24
  Copying 54
  Persistence 36

T

Title method 89