OBJECTSTORE

RAPID DATABASE DEVELOPMENT FOR JAVA

RELEASE 3.0

October 1998

ObjectStore Rapid Database Development for Java

Release 3.0, October 1998

ObjectStore, Object Design, the Object Design logo, LEADERSHIP BY DESIGN, and Object Exchange are registered trademarks of Object Design, Inc. ObjectForms and Object Manager are trademarks of Object Design, Inc.

All other trademarks are the property of their respective owners.

Copyright © 1989 to 1998 Object Design, Inc. All rights reserved. Printed in the United States of America. Except as permitted under the Copyright Act of 1976, no part of this publication may be reproduced or distributed in any form or by any means, or stored in a database or retrieval system, without the prior written permission of the publisher.

COMMERCIAL ITEM — The Programs are Commercial Computer Software, as defined in the Federal Acquisition Regulations and Department of Defense FAR Supplement, and are delivered to the United States Government with only those rights set forth in Object Design's software license agreement.

Data contained herein are proprietary to Object Design, Inc., or its licensors, and may not be used, disclosed, reproduced, modified, performed or displayed without the prior written approval of Object Design, Inc.

This document contains proprietary Object Design information and is licensed for use pursuant to a Software License Services Agreement between Object Design, Inc., and Customer.

The information in this document is subject to change without notice. Object Design, Inc., assumes no responsibility for any errors that may appear in this document.

Object Design, Inc.
Twenty Five Mall Road
Burlington, MA 01803-4194

Contents

Preface 5

Chapter 1 Overview 1

Introducing the Database Designer 2
Creating Relationships. 5
Viewing the Database Design 6
Introducing the Component Wizard. 8
Generating Project Files 10

Chapter 2 Design the Database 13

Starting the Database Designer. 14
Creating Classes and Fields. 15
Creating Accessor Methods 18
Creating Relationships. 20
  Create an Inheritance Relationship 20
Enhancing the Design 22
  Create Two Additional Classes 22
  Create Two Additional Relationships 23
  Create an Additional Method for **Employee**. 24
Reviewing the Completed Database Design 27
  Save the Database Design 27

Chapter 3 Run the Component Wizard. 29

Starting the Component Wizard 30
  Start the Component Wizard from Visual J++ 30

Start the Component Wizard from the Database Designer . . 31
Selecting the Database and Classes 33
Select a Database Design (Visual J++ Only) 33
Select Classes 33
Defining Database Entry Points 34
Create Roots for Top-Level Classes. 34
Define Database Roots 35
Display New Package Information 35
Examining the Component Wizard Code. 36
Class Definitions. 36
Class Constructor 37
Class Extent Handling 37
The **extents.Java** file 38
Defining Methods 41
Writing an Application 43
The **top.Java** class. 43
Writing the Main Function 44
Building the Project 46
Building the Project with JDK 46
Building the Project Using Visual J++ 1.1. 48

Preface

Purpose

ObjectStore Rapid Database Development for Java is a tutorial that shows you how to build an ObjectStore application — from database design to code generation — using two new Rapid Database Development (RDD) tools: the Database Designer and the Component Wizard.

This tutorial shows you how to

- Use the Database Designer to design a database
- Use the Component Wizard to generate Java classes that form the skeleton of your application
- Implement a simple Java application using code created by the Component Wizard

Audience

This tutorial is for Java application developers who are new to using ObjectStore. In addition to describing productivity tools, it introduces some basic concepts.

Scope

This tutorial supports ObjectStore for Java, Release 3.0 or later.

How This Tutorial Is Organized

This tutorial contains the following chapters:

- Chapter 1, Overview, on page 1, introduces the two productivity tools that help you design and generate code for ObjectStoreapplications — the Database Designer and the Component Wizard — and describes the application that you create as you work through the tutorial.
- Chapter 2, Design the Database, on page 13, describes how to use the Database Designer's graphical user interface to quickly create a database design to support your application.
- Chapter 3, Run the Component Wizard, on page 29, describes how to use the Component Wizard to create Java classes and takes a look at some of the Wizard-generated code. It also shows how to write a simple Java application on top of the Component Wizard-generated code.

Software Requirements

To run both the Database Designer and the Component Wizard, you will need the following:

- ObjectStore for Java, Release 3.0 or later.
- Any supported Java implementation. See the **readme.htm** file in the ObjectStore for Java installation directory for a complete list.

Notation Conventions

This book uses the following conventions:

Convention	***Meaning***
Bold	Bold typeface indicates user input or code.
Sans serif	Sans serif typeface indicates system output.
Italic sans serif	Italic sans serif typeface indicates a variable for which you must supply a value. This most often appears in a syntax line or table.
Italic serif	In text, italic serif typeface indicates the first use of an important term.
[ ]	Brackets enclose optional arguments.
{ *a* | *b* | *c* }	Braces enclose two or more items. You can specify only one of the enclosed items. Vertical bars represent OR separators. For example, you can specify *a* or *b* or *c*.
...	Three consecutive periods indicate that you can repeat the immediately previous item. In examples, they also indicate omissions.

Internet Sources of More Information

Object Design

Object Design's site on the World Wide Web is the source for company news, white papers, and information about product offerings and services. Point your browser to **http://www.objectdesign.com/** for more information.

Other ObjectStore products

Object Design offers a comprehensive set of rapid development and enterprise integration tools. For information about these and other Object Design products, point your browser to **http://www.objectdesign.com/products/products.html**.

Product support

Object Design's support organization provides a number of information resources and services. Their home page is at **http://support.objectdesign.com/**. From the support home page, click Tech Support Information to learn about support policies, product discussion groups, and the different ways Object Design can keep you informed about the latest release information — including the web, **ftp**, and email services.

Training

You can obtain information about training courses from the Object Design web site (**http://www.objectdesign.com**). From the home page, select **Services** and then **Education**.

Customers in North America can obtain information about Object Design's educational offerings by calling 781.674.5000, Monday through Friday from 8:30 AM to 5:30 PM Eastern Time. You can also get up-to-date information and register for education courses at **http://www.objectdesign.com/services/services.html.**

Your Comments

Object Design welcomes your comments about ObjectStore documentation. Send your feedback to **support@odi.com**. To expedite your message, begin the subject with **Doc:**. For example:

Subject: Doc: Incorrect message on page 76 of reference manual

You can also fax your comments to 781.674.5440.

Chapter 1
Overview

This chapter introduces you to two new Rapid Database Development (RDD) tools: the Database Designer and the Component Wizard.

This tutorial is designed to give you hands-on experience with the Database Designer and Component Wizard. Try the exercises in the tutorial to see how quickly and easily you can develop applications using ObjectStore. The tutorial, which starts in Chapter 2, Design the Database, on page 13, assumes that you have installed ObjectStore and these RDD tools.

In this chapter

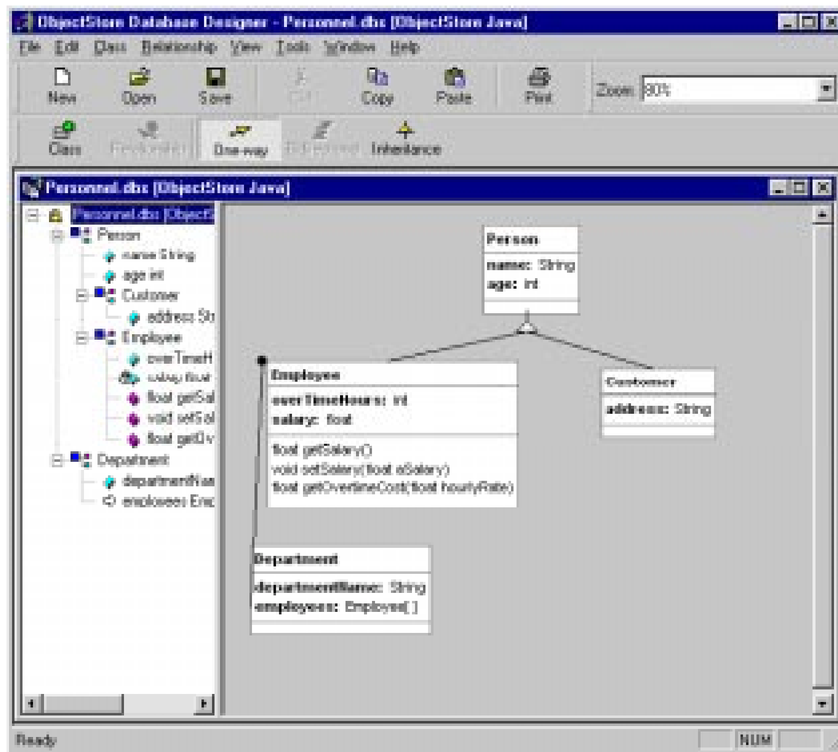
This chapter covers the following topics:

Introducing the Database Designer	2
Introducing the Component Wizard	8
Creating Relationships	5
Viewing the Database Design	6
Generating Project Files	10

Introducing the Database Designer

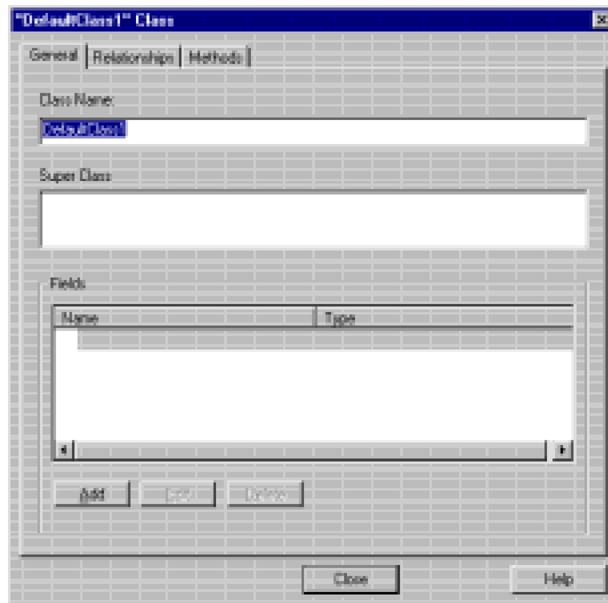
The Database Designer is a tool used for designing PSE Pro databases. It provides a graphical user interface (GUI) that lets you quickly create all the elements you need to represent your database design:

- Classes
- Fields
- Methods
- Relationships



You can define database elements by simply clicking a button on the Database Designer toolbar.

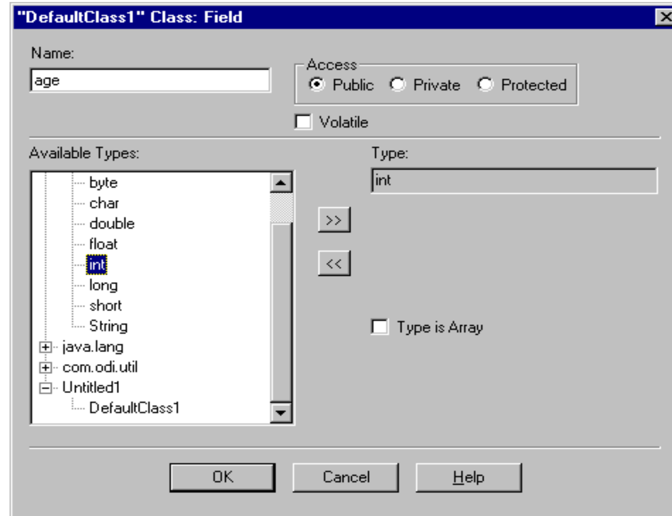
For example, clicking the Class button opens the Class dialog box from which you can create a class, its fields, and methods.



The Database Designer provides support for both PSE Pro and primitive Java data types.

Design from the top down or bottom up

You can design your database using either the top-down or bottom-up approach. For example, you can quickly sketch a database design, populate it with only the classes you intend to implement, and then change the Database Designer default names as needed. You can create a class and later fill in the details such as fields, methods, and relationships. Or you can define each class in detail, one at a time, using property sheets in the Class dialog box.



Creating Relationships

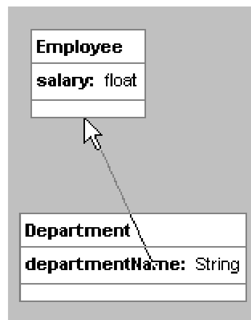
You can easily create a relationship (*one-way* or *inheritance*) between classes.

To create a relationship between classes,

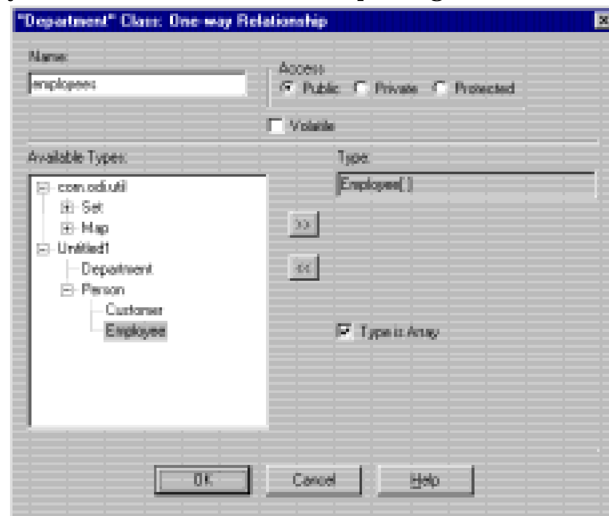
1 Select the relationship type you want to create by clicking the corresponding button on the Database Designer toolbar.

2 Press and hold the Shift key.

3 Click and drag the mouse pointer from one class to the corresponding class you are creating the relationship with, and then release the mouse button.



The Database Designer opens a dialog box with default values for the relationship you created. You can accept the default values or you can define the relationship using actual names.



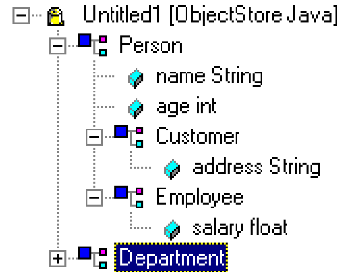
Viewing the Database Design

You can view the information in your database design in three ways:

- Class hierarchy view
- Database diagram
- Element dialog boxes

Class hierarchy

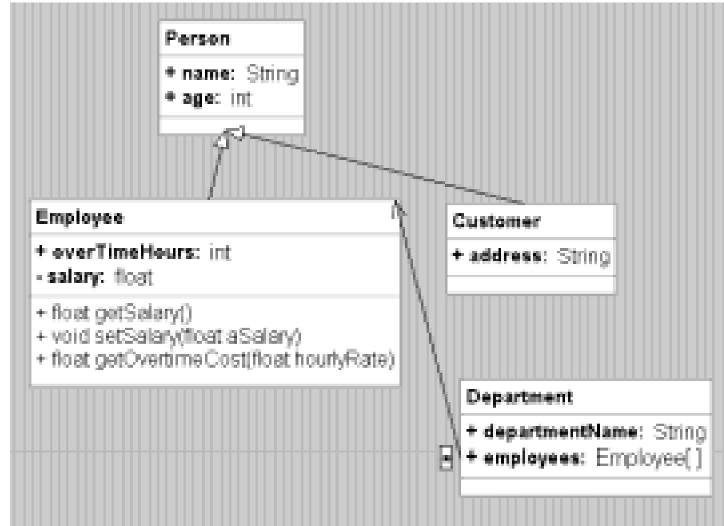
The *class hierarchy* view displays a detailed view of all the elements in your database design.



This view is updated automatically whenever you make a change to the database design, such as adding a field to a class, creating a relationship, or changing a default name to one appropriate for your design. Color-coded symbols help you quickly identify different design elements.

Database diagram

The *database diagram* provides a higher-level view of the database design by showing which classes are related and how. Note that you can choose to display either of two styles of standard modeling notation — OMT (object modeling technique) or UML (unified modeling language). This tutorial uses UML notation.



Database diagrams can be printed and are useful for documenting your design. The design diagram is an effective way to convey database information to a nontechnical audience.

Element dialog boxes

Each class and relationship has an associated dialog box that contains detailed information. To view the relationship, double-click an element in the design diagram, or select Properties from the Class or Relationship menu bar.

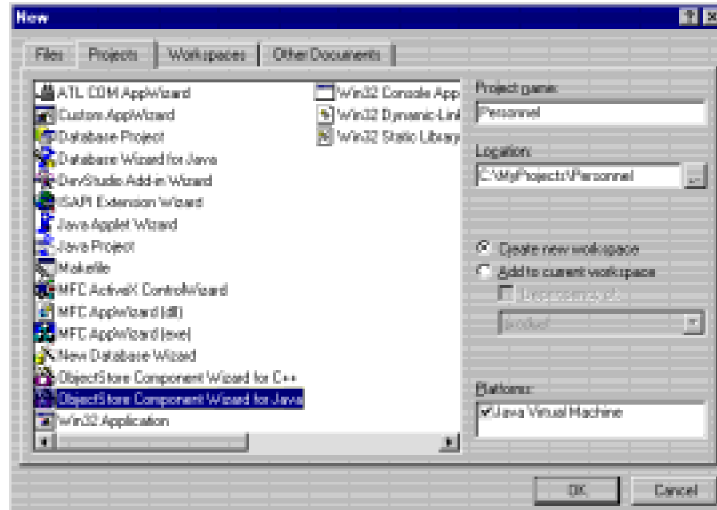
The Class dialog box includes a separate property sheet for creating and maintaining each of the following database elements:

- Fields
- Methods
- Relationships

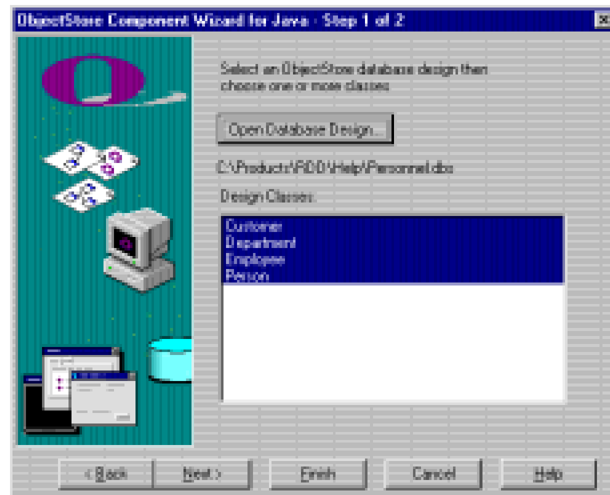
Introducing the Component Wizard

The Component Wizard is a code generator that uses the database design file you create with the Database Designer to generate ObjectStoreclasses.

You can launch the Component Wizard directly from Visual J++.



If you are using another Java compiler, you can start the Component Wizard from the Tools menu in the Database Designer.



You can select a database design and choose one or more classes, and then click Next to step through the creation process, or you can use the default values provided by the Component Wizard and then click Finish.

Generating Project Files

The Component Wizard automatically generates several types of files.

Name	Size	Type	Modified
cfpargs	1KB	File	8/28/98 12:21 AM
Customer.java	1KB	Java Source File	8/28/98 12:21 AM
Department.java	2KB	Java Source File	8/28/98 12:21 AM
doall.bat	1KB	MS-DOS Batch File	8/28/98 12:21 AM
Employee.java	2KB	Java Source File	8/28/98 12:21 AM
Ext_OSTreeSet.java	1KB	Java Source File	8/28/98 12:21 AM
Ext_OSVector.java	1KB	Java Source File	8/28/98 12:21 AM
Extents.java	2KB	Java Source File	8/28/98 12:21 AM
Person.java	2KB	Java Source File	8/28/98 12:21 AM
Personnel.dsp	4KB	Project File	8/28/98 12:21 AM
Personnel.dsw	1KB	Project Workspace	8/28/98 12:21 AM
Personnel.ncb	41KB	NCB File	8/28/98 12:21 AM
Personnel.opt	48KB	OPT File	8/28/98 12:21 AM
Top.java	1KB	Java Source File	8/28/98 12:21 AM

The following table describes some of the important files created by the Component Wizard.

File Name	***Description***
class.**class**	Class definition for every class in the database design. Class files also contain the code for the CRUD (create, read, update, and delete) functions associated with each class.
project.**dsw**	Workspace file, which bundles project and custom build dependencies (like a makefile for other makefiles). Generated only if you are using Visual J++.
doall.bat	A batch file that compiles the Java classes, calls the ObjectStorepostprocessor, and runs the application.
cfpagrs	Contains the parameters for the postprocessor.
extents.java	A *wrapper* around a container class that manages extents by automatically associating the container class with the proper root.
top.java	A Java class that contains the skeleton of a “main” function that you can use as a foundation for writing your ObjectStoreapplication.

Now that you have been introduced to the Database Designer and the Component Wizard, you can begin using these tools to design, write, and build your applications.

Chapter 2
Design the Database

This chapter shows you how to use the Database Designer to define a database for a simple personnel application.

This chapter covers the following topics:

Starting the Database Designer 14

Creating Classes and Fields 15

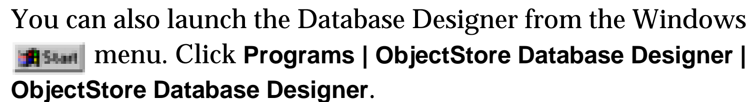
Creating Relationships 20

Enhancing the Design 22

Reviewing the Completed Database Design 27

Starting the Database Designer

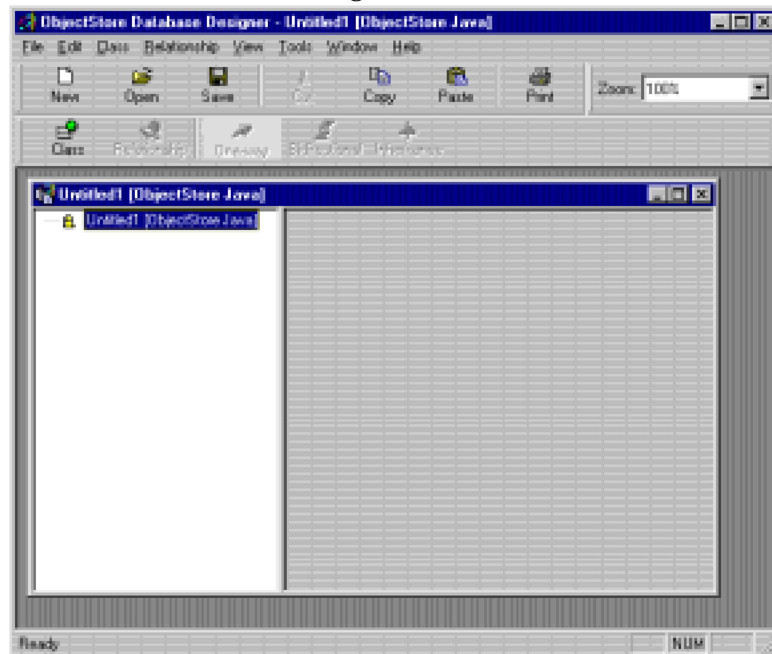
When you start the Database Designer, you need to specify the type of schema you want to design. To start the Database Designer, double-click the Database Designer icon located in the ObjectStore program group that was created when you installed ObjectStore.



You can also launch the Database Designer from the Windows Start menu. Click **Programs | ObjectStore Database Designer | ObjectStore Database Designer**.

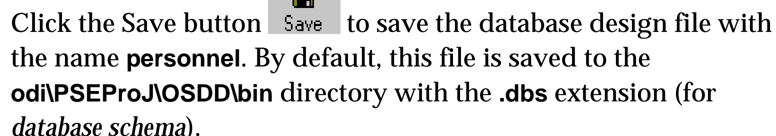
Open a workspace

To open a new database design file, or *workspace*, click the New button on the Database Designer toolbar.



The Database Designer opens an unnamed, empty workspace where the classes, fields, methods, and relationships that represent the schema of your database are stored.

Save the empty workspace



Click the Save button to save the database design file with the name **personnel**. By default, this file is saved to the **odi\PSEProJ\OSDD\bin** directory with the **.dbs** extension (for *database schema*).

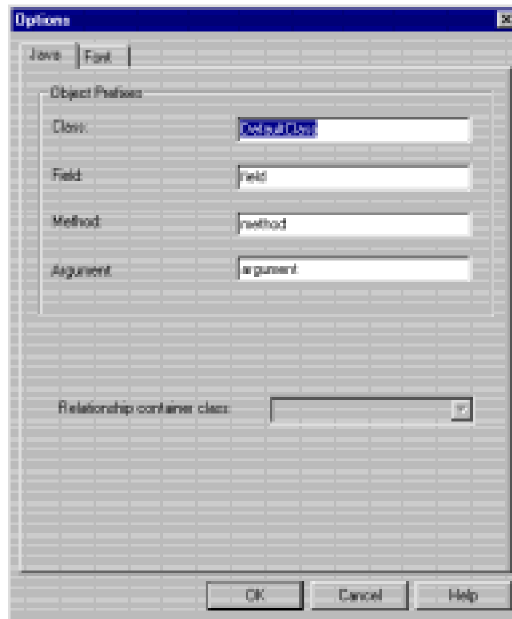
Creating Classes and Fields

After you open a new workspace, you can create the classes and fields that will become the foundation of your **personnel** database.

Check default names

The Database Designer uses default prefixes for class and field names. You can use these default values to quickly sketch a design and then fill in the details later.

To open the Options dialog box and view the default values, click **Tools | Options**.

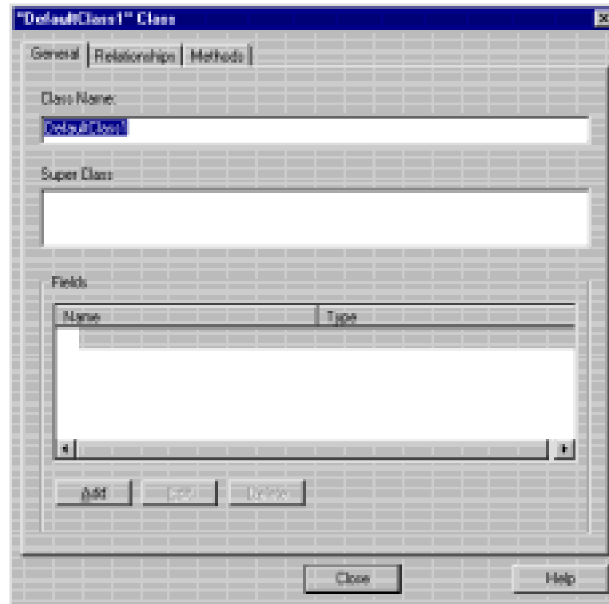


Create a **Person** class

Use the Class dialog box to create and edit classes, and to define methods and relationships.

To create a class called **Person:**

1 Click the Class button on the Database Designer toolbar, or click **Class | Add** on the menu bar to open the Class dialog box.



2 In the General property sheet, change the default value in the Class Name field from **DefaultClass1** to **Person**.

Add Name and Age fields

3 Add the following fields to the **Person** class. Each field uses a primitive data type, therefore you can add your field information directly in the Class dialog box:

Field	***Type***
name	**String**
age	**int**

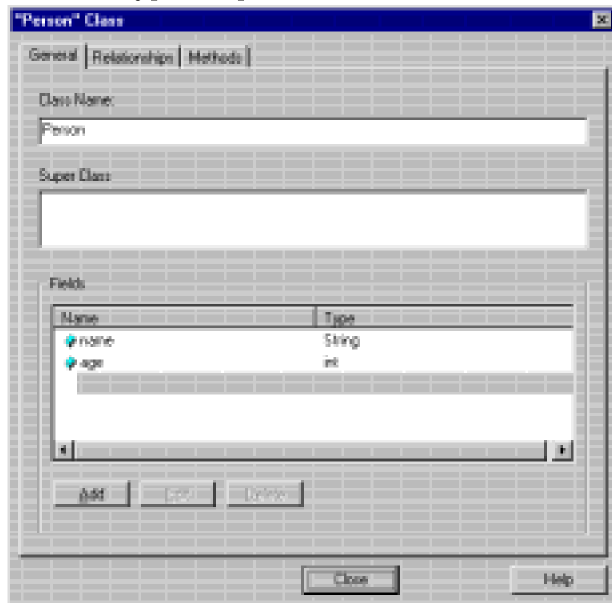
You can also add fields to your classes at a later time.

4 Click a row in the Name column of the Fields list box and then type the field name.

5 Click a row in the Type column to display a list of primitive data types.

Tip: You can access Java, ObjectStore, and user-defined types from the Field Relationship dialog box. To display this dialog box, click the ellipsis entry (...) in the Type drop-down list, or click the Add button.

6 Select the type and press Enter to define the field.



Create an **Employee** class

Create an **Employee** class using the same procedure you used to create the **Person** class. Then define the following two fields:

Field	***Type***
salary	**float**
OvertimeHours	**int**

Note: To make the salary field **private**, click the **Private** option button located in the Access group box of the corresponding Relationship dialog box.

Creating Accessor Methods

After you create classes, you can create accessor methods for each class. This example creates accessor methods for the **Employee** class. These methods will return and set the value of the salary field.

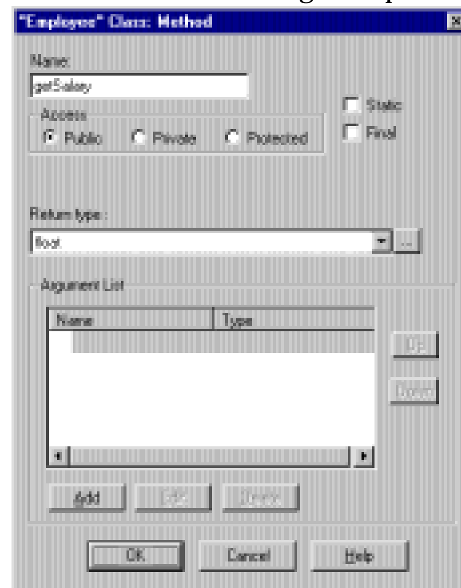
The **get** method

To create a method that returns the value of the salary field:

1 Click the Methods tab of the Class dialog box.

2 Click the Add button.

The Class: Method dialog box opens.



3 Type **getSalary** in the Name field.

4 Select **float** from the Return type list and then click OK.

The **set** method

To create a method to set the value of the salary field:

1 Click the Add button on the Methods tab of the Class dialog box.

The Class: Method dialog box opens.

2 Type **setSalary** in the Name field.

3 Select **void** from the Return type list box.

4 Click the Name field in the Argument list box to display the default argument name.

5 Change the default argument name by typing **Salary** in the Name field.

6 Select **float** from the Return type list and then click OK.

The next section shows you how to use the database diagram to create a relationship between the **Person** class and the **Employee** class, and then create a method for the **Employee** class.

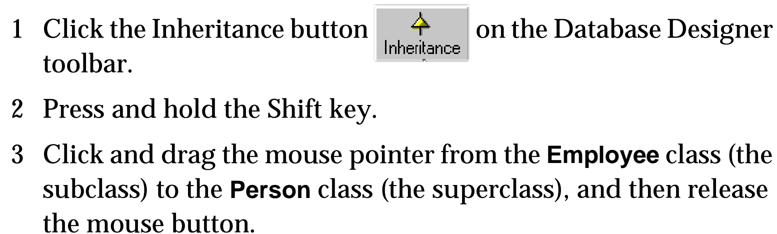
Creating Relationships

You now have two classes defined for your **personnel** database, the **Person** class and the **Employee** class. In this section, you will create an *inheritance* relationship between these two classes using a database diagram. The following database diagram shows the contents of these two classes.

Create an Inheritance Relationship

An inheritance relationship consists of a *superclass* and a *subclass*; the subclass inherits attributes from the superclass. Inheritance relationships are sometimes called *is-a relationships*, as in, an **Employee** *is a* **Person**.

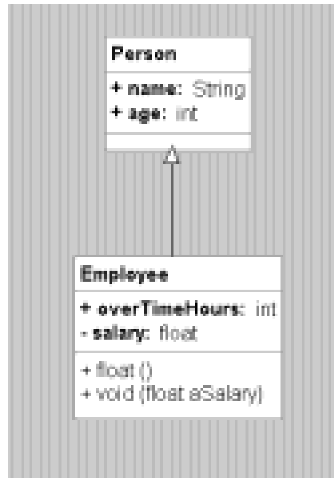
To create an inheritance relationship between the **Employee** and **Person** classes:

1. Click the Inheritance button  on the Database Designer toolbar.
2. Press and hold the Shift key.
3. Click and drag the mouse pointer from the **Employee** class (the subclass) to the **Person** class (the superclass), and then release the mouse button.

   Note that when you define an inheritance relationship graphically, you must always identify the subclass first.

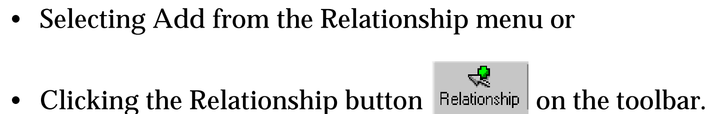
Tip: You can drag a class anywhere within the database diagram.

When the relationship is created, a line that connects the two classes appears in the diagram.



Note: This example uses UML notation, which uses an arrow symbol at the end of the line to identify the inheritance relationship. OMT notation uses a triangle symbol instead of an arrow.

You can also create a relationship by selecting a class name in the diagram and then either

- Selecting Add from the Relationship menu or
- Clicking the Relationship button  on the toolbar.

Enhancing the Design

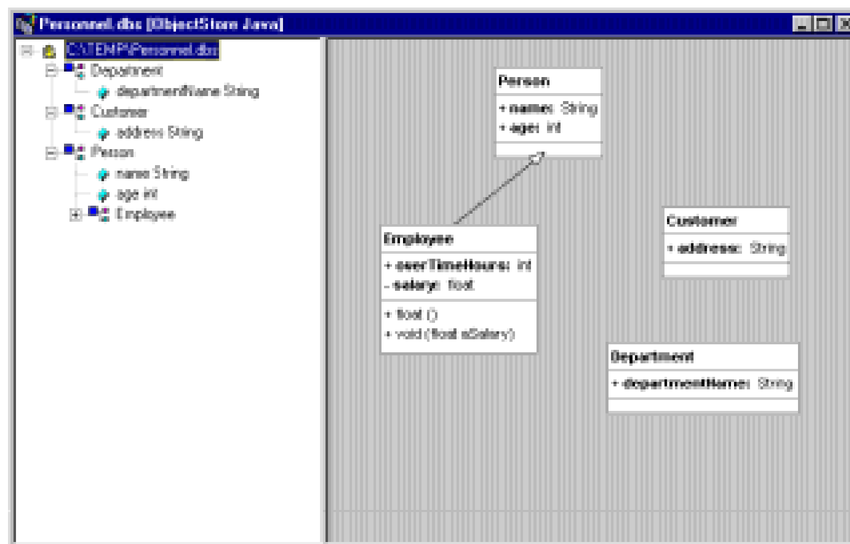
You now have defined an inheritance relationship between the **Employee** and **Person** classes in your **personnel** database design. In this section, you add more classes, relationships, and a method to make the design more complete.

Create Two Additional Classes

Add two classes, **Customer** and **Department**, to your database design. The field name and type for each class are defined in the following table:

Class	*Field Name, Type*
Customer	**code**, **int**
Department	**departmentName**, **String**

After you add these classes, your database diagram will look similar to the following diagram:



Tip: To get started, click the Class button on the toolbar, or select Add from the Class menu. If you need more help, see Create a Person class on page 16.

Create Two Additional Relationships

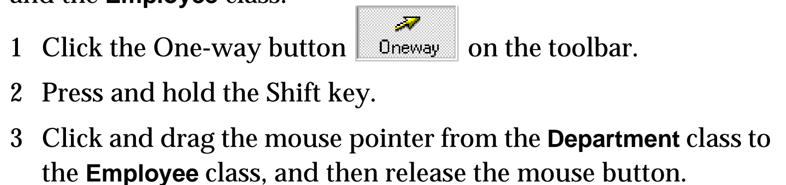
Inheritance

Now add an inheritance relationship between the **Customer** and **Person** classes. Click the Inheritance button on the Database Designer toolbar, press and hold the Shift key, and then drag the mouse pointer from the **Customer** class to the **Person** class. (See page 20 if you need more specific instruction.)

One-way

Next, create a one-way relationship between the **Department** and **Employee** classes. Although you can create any relationship graphically, some relationships require you to provide additional information before the relationship can be created.

To create a one-way relationship between the **Department** class and the **Employee** class:

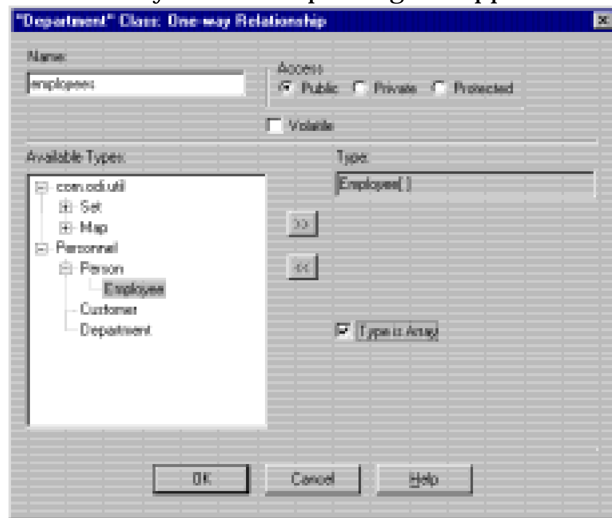
1 Click the One-way button 

on the toolbar.

2 Press and hold the Shift key.

3 Click and drag the mouse pointer from the **Department** class to the **Employee** class, and then release the mouse button.

The One-way Relationship dialog box appears.



4 Click in the Name field and then change the default field name to **employees**.

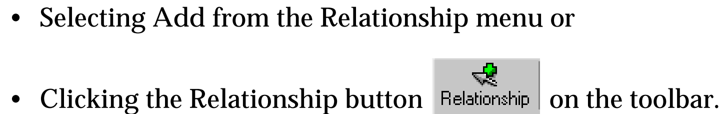
5 In the Available Types box, double-click on **Employees**.

6 Click the Type is Array check box.

7 Click OK.

This defines a one-to-many relationship between **Department** and **Employee**; in other words, a department contains zero to many employees. In this example, the relationship is implemented using a Java array.

You can also create a relationship by selecting a class name in the diagram and then either

- Selecting Add from the Relationship menu or
- Clicking the Relationship button  on the toolbar.

Create an Additional Method for Employee

Now you create another method for the **Employee** class. Consider that the end user of your **personnel** application will need to determine the amount of overtime costs generated by a particular employee.

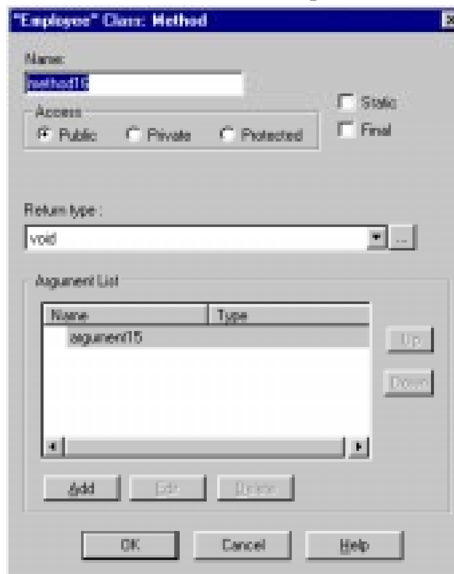
You need to add a method declaration to the **Employee** class in your database design and then write a simple method definition using Visual J++ or another code editor.

To create a method for **Employee**:

1 Double-click the Employee class name in the database diagram to open the Class dialog box.

2 Click the Methods tab to display the Methods property sheet.

3 Click the Add button to open the Class: Method dialog box.



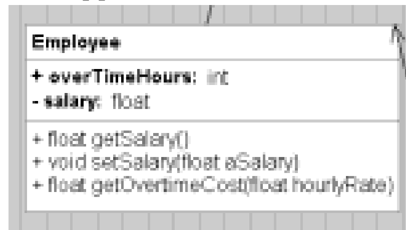
4 Click the Name field and type the name **getOvertimeCost**.

5 Select **float** from the Return Type list.

6 Click the Name field in the Argument list box and change the default argument name to **hourlyRate**.

7 Select **float** from the Type drop-down list and then click OK.

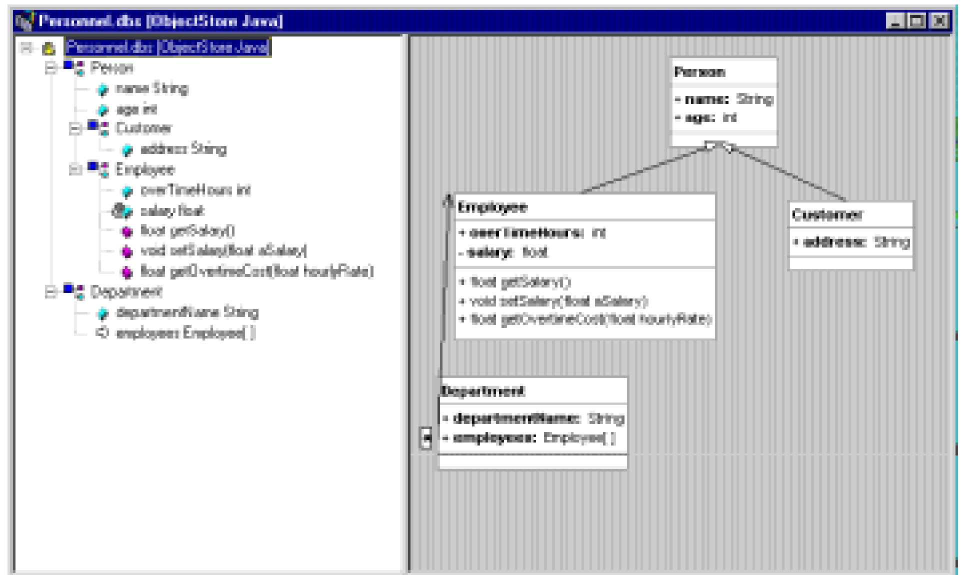
The new method appears on the Methods property sheet, and also appears in the **Employee** class in the database diagram.



8 Click Close to close the Class dialog box.

Reviewing the Completed Database Design

The completed design diagram for the **personnel** database should include all the classes, fields, and relationships shown in the following figure. Note that the placement of individual classes might vary — you can arrange the classes in a database diagram according to your preferences.



Save the Database Design

When you are satisfied that your design is correct, save the design file. This file is used as input to the ObjectStore Component Wizard, which is described in the following chapter.

Chapter 3
Run the Component Wizard

This chapter shows you how to use the Component Wizard to generate the code you will use to implement your application. Using the **personnel** database design file you created in Chapter 2 as input, you can select a number of code-generation options and let the Component Wizard do the rest.

This chapter covers the following topics:

Starting the Component Wizard 30

Selecting the Database and Classes 33

Defining Database Entry Points 34

Examining the Component Wizard Code 36

Writing an Application 43

Building the Project 46

Starting the Component Wizard

The Component Wizard is available as a Visual J++ plug-in or as a stand-alone tool that you can launch from the Database Designer. This chapter describes both ways to start the Component Wizard.

Visual J++ plug-in

Use the Component Wizard from within Visual J++ to automatically generate the Java classes for your application. The compiler uses the project information in order to be aware of all the files in your application, as well as the code that causes the compiler to invoke the postprocessor on all your persistent classes. You can select any database design from which to generate code.

Stand-alone tool

Use the Component Wizard as a stand-alone tool if you are using a Java compiler other than Visual J++. The code generated by the Component Wizard is based on the active design in the Database Designer.

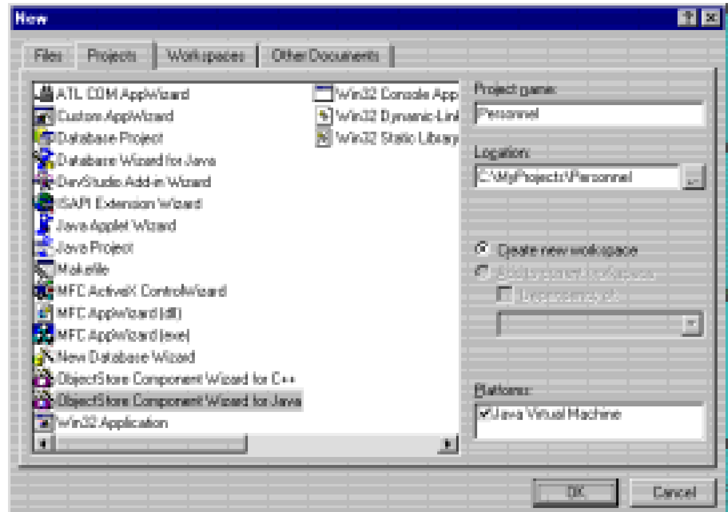
Start the Component Wizard from Visual J++

To start the Component Wizard from Visual J++:

1 Start Microsoft Developer Studio.

2 Click **File | New** to open the New dialog box.

3 Click the Projects tab and select **ObjectStore Component Wizard for Java** in the Projects list.



4 Type the name **Personnel** in the Project Name field.

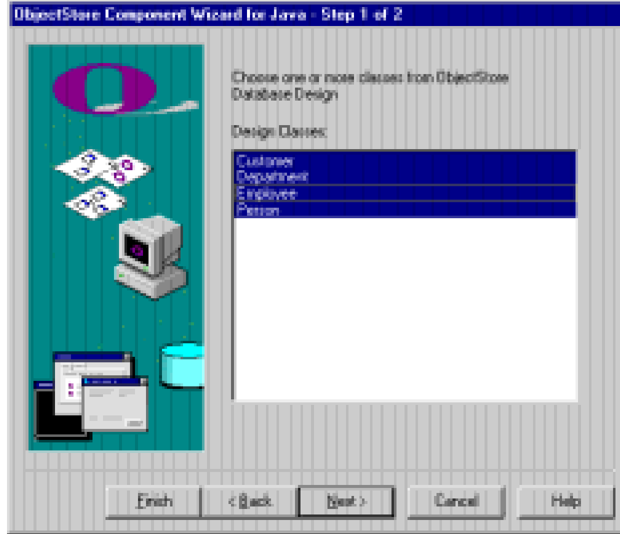
5 If necessary, change the directory specified in the Location field. This field specifies where the Component Wizard will write the files it creates.

6 Accept the default values for the remaining fields and then click OK.

Start the Component Wizard from the Database Designer

To start the Component Wizard from the Database Designer, select **Component Wizard** from the Tools menu.

After you start the Component Wizard, the ObjectStore Component Wizard for Java Step 1 of 2 dialog box opens.



Every dialog box of the Component Wizard has a Next and a Finish button. Both buttons advance the Component Wizard.

The Next button

Click Next to let the Component Wizard guide you through one step at a a time. You should consider using Next so that you can become familiar with all of the Component Wizard's features.

The Finish button

Click Finish to let the Component Wizard complete automatically, using default values wherever possible. Note that the Finish button might be unavailable if a subsequent dialog box in the Component Wizard has one or more fields that require user input.

Selecting the Database and Classes

If you are running the Component Wizard from within Visual J++, you first need to select a database. Then select the classes that you want to use in your application.

If you are running the Component Wizard from the Database Designer, the active database will be used. Then you need to select the classes that you want to use in your application.

Select a Database Design (Visual J++ Only)

To select a database design:

1 Click the Open Database Design button to display the Open window.

2 Locate and open the database design file for the **personnel** database you created in Chapter 2.

Tip: ObjectStore database design files have a default extension of **.dbs**.

Select Classes

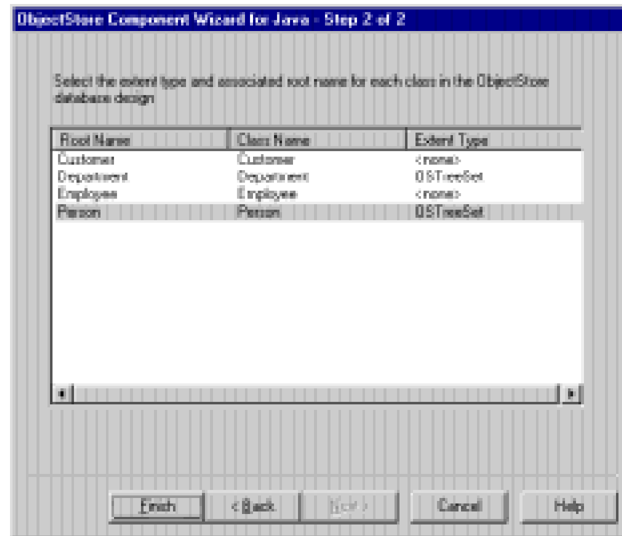
When you start the Component Wizard, all the classes in the database appear by default in the Design Classes list box.

Click Next to continue to the next step, as described in Defining Database Entry Points.

Tip: After you become more familiar with the Component Wizard, you can click the Finish button whenever it is available.

Defining Database Entry Points

Now you need to define the entry points, called *roots*, of your database. A database root gives an object a *persistent name*, which allows the object to serve as an entry point into *persistent memory*. When an object has a persistent name, any process can look it up by that name and retrieve it; you can find related objects by navigating from object to object.



Create Roots for Top-Level Classes

A *database root* can be any type of object or collection of objects. By default the Component Wizard creates a root for each top-level class in a schema that points to one of the collection types for ObjectStore. These types include **OSTreeSet** and **OSVector**. The Component Wizard uses **OSTreeSet** by default. For each of the selected classes, the Component Wizard generates the code to create the root, create the container, and update the class extent, that is, all the persistent instances of the class.

For more information

To learn more about roots, entry-point objects, and collections, see the *ObjectStoreJava API User Guide*.

Define Database Roots

The Component Wizard displays the default values that will be used to define the database roots for your application.

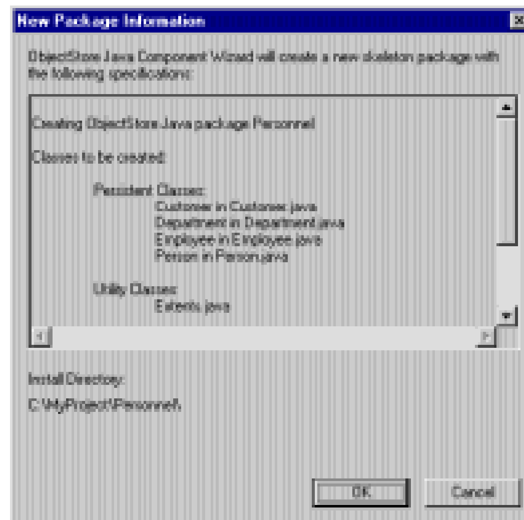
You can change these defaults by

- Changing the root name — the default root name is the same as that of its class.
- Removing a root — each class that is defined in the database becomes a database root by default unless you specify **None** as the extent type.
- Changing the extent type — each root is defined by default as an instance of **OSTreeSet**.

This tutorial uses the default values supplied by the Component Wizard.

Display New Package Information

When you click the Next button, the Component Wizard displays the New Package Information dialog box. Click OK to continue.



You are now ready to build an application.

Examining the Component Wizard Code

This section examines some of the code generated by the Component Wizard.

Class Definitions

For each class you define in the Database Designer, the Component Wizard generates a file called *ClassName***.java** where *ClassName* is the name of the class. The *ClassName***.java** file contains the class definition, including attributes, declarations, and method signatures.

The following is an excerpt from the **Employee.java** file:

```
public class Employee extends Person{
// Attributes
   public int overTimeHours;
   private float salary;

...

// Operations
   public  float getSalary()
   {
      float ret=0;
      //TODO: Add your code here
      return ret;
   }
   public  void setSalary(float aSalary)
   {
      //TODO: Add your code here
   }
   public  float getOvertimeCost(float hourlyRate)
   {
      float ret=0;
      //TODO: Add your code here
      return ret;
   }
...

}
```

You need to provide the method body for each method you define in the Database Designer. See Defining Methods on page 41 for more information on writing methods.

Class Constructor

For each class, the Component Wizard generates an empty constructor with no parameter, and a constructor that initializes all the attributes to the values passed as parameters to the constructor.

The following is an excerpt from the **Person.java** file:

```
public class Person{
...

    ////////////////////////////////////////////////////////
    // Constructor
    public Person()
    {
    }

    public Person(String _name, int _age)
    {
        name=_name;
        age=_age;
    }
...
}
```

Class Extent Handling

For each class for which an extent has been defined, the Component Wizard generates the following attribute definition:

```
//Extents
    public static Ext_OSTreeSet Ext = new Ext_OSTreeSet("Person");
```

The **Ext** attribute points to the extent for the class that is implemented as

Ext_OSTreeSet

The Component Wizard also generates the following method:

```
public void preFlushContents()
    {
        Segment theSegment=
Session.getCurrent().segmentOfpreFlushContentsObject();
        Database db = theSegment.getDatabase();
        Collection theExtent = (Collection) Ext.getExtents(db);
        if (!theExtent.contains(this))
            updateExtents(db, true);
}
```

The **preFlushContents** method is called when the instance changes its persistent state. The previous implementation of the **preFlushContents** method ensures that when the instance becomes persistent because it has been included or referred to by another persistent object, it gets inserted into the proper extent.

The following generated method explicitly inserts an instance into its extent:

```
void updateExtents(Database db, boolean add)
   {
      Ext.update(this, db, add);
}
```

This method should be called after an instance has been created to insert it into the proper extent. When inserted into a persistent extent, the newly created instance becomes persistent itself.

For more information

To learn more about persistence by reachability, see the *ObjectStoreJava API User Guide*.

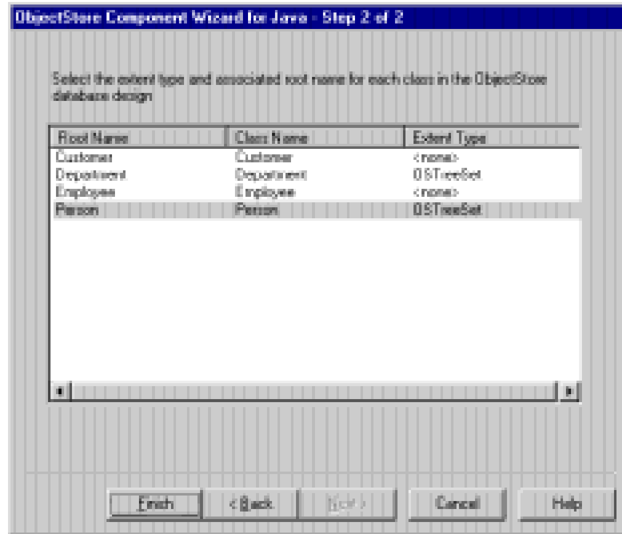
The extents.Java file

The **extents.Java** file contains the **Extents** class, which implements the default extent-handling functionality.

An *extent* is a collection of all the instances of a class in the database and can be implemented using any container class. The Component Wizard allows you to implement extents using either an **OSTreeSet** or **OSVector**; that is why the **Extents** class is an abstract class that gets specialized by **Ext_OSTreeSet** and **Ext_OSVector**. These classes are the classes for which an extent-handling mechanism has been specified in the Component Wizard.

Working with extents

To enable an application to retrieve an extent, the collection has to be reachable by navigating a root, that is, a named entry point into persistent memory. You can specify both the name of the root and the type of the collection used to implement a class extent in the ObjectStore Component Wizard for Java Step 2 of 2 dialog box.



In the **personnel** database, you implemented the extent of the **Department** and **Person** classes using an **OSTreeSet**; therefore, the class **Ext_OSTreeSet** is used. By default, root names are based on class names.

```
public abstract class Extents
{
    String m_name;
    Database m_db=null;
    Object m_obj;
    Object m_root;

    public String getRootName() {return m_name;};

    public Extents(String rootname)
    {
        m_name = rootname;
    }

    public abstract Object createExtents(Database db);

    public Object getExtents(Database db)
    {
        try{
```

```
            if (db!=null){
                m_db= db;
                m_root= m_db.getRoot(m_name);
            }
        }
        catch(DatabaseRootNotFoundException RootNotFound){
            m_root = null;
        }

        if (m_root==null){
            m_root = createExtents(db);
            setExtents(m_root);
        }

        return m_root;
    }

    public void update(Object obj, Database db)
    {
        m_obj= obj;
        m_db= db;
    }

    protected void setExtents(Object root)
    {
        m_root = root;
        m_db.createRoot(m_name, root);
    }

    protected boolean checkType(Object obj, String type)
    {
        Class c = obj.getClass();
        String name = c.getName();
        return (name.compareTo(type) == 0);
    }
}
```

Public methods

The public methods of this class are

public String getRootName()

Returns the name of the root that points to the extent.

public abstract Object createExtents(Database db);

Implemented by the **Ext_OSTreeSet** and **Ext_OSVector** classes. It creates the proper container collection for the extent.

public Object getExtents(Database db)

Returns the class extent. If the extent does not exist, it calls the **createExtents** method to create one and attaches it to the proper root.

public void update(Object obj, Database db)

Inserts the **obj** object into the extent.

For more information

For more information on database roots, see the *ObjectStoreJava API User Guide*.

Defining Methods

The Component Wizard generates the signatures of the methods you define with the Database Designer; you will need to define the method implementation.

Consider the following example of the body of the **getSalary** method of the **Employee** class.

The Component Wizard generates the following code:

```
public  float getSalary()
{
    float ret=0;
    //TODO: Add your code here
    return ret;
}
```

The **getSalary** method returns the value of the **salary** attribute. The method body might look like this:

```
public  float getSalary()
    {
        return salary;
    }
```

The **setSalary** method sets the value of the **salary** attribute. The method body might look like this:

```
public  void setSalary(float aSalary)
   {
      //TODO: Add your code here

      salary = aSalary;

   }
```

The **getOvertimeCost** method should return the product of the **hourlyRate** parameter and the number of overtime hours stored in the **overTimeHours** attribute:

```
   public  float getOvertimeCost(float hourlyRate)
   {
      float ret=0;
      //TODO: Add your code here
      ret= overTimeHours * hourlyRate;
      return ret;
   }
```

Writing an Application

This section describes how to get started writing an ObjectStoreapplication using the files generated by the Component Wizard.

The top.Java class

The **top.java** class contains the skeleton of a simple **main** for your application. You can use this class to get started with the implementation of your application.

```
final public class Top
{
    public static Database db=null;

    public static void main(String argv[]) throws java.io.IOException
    {
        String dbName="db.odb";
        ObjectStore.initialize(null, null);

    try{
        try{
            db = Database.open(dbName,  ObjectStore.OPEN_
UPDATE);
        }
        catch (DatabaseNotFoundException e){
            db = Database.create(dbName,ObjectStore.ALL_READ |
ObjectStore.ALL_WRITE);
        }
        Transaction t = Transaction.begin(ObjectStore.UPDATE);

        //insert your code here


        t.commit();
        db.close();
    }
    catch(Exception e){
        System.out.println(e);
    }
    finally{
        ObjectStore.shutdown(true);
    }
  }
}
```

The **db** attribute points to a **Database** object:

```
public static Database db=null;
```

The database gets opened in the main function:

```
try{
    db = Database.open(dbName,  ObjectStore.OPEN_UPDATE);
}
```

If it does not exist it is created:

```
catch (DatabaseNotFoundException e){
    db = Database.create(dbName,ObjectStore.ALL_READ |
ObjectStore.ALL_WRITE);
}
```

The main function also shows how you can start and commit a transaction:

```
Transaction t = Transaction.begin(ObjectStore.UPDATE);
//insert your code here
t.commit();
```

Writing the Main Function

In this example, you create a couple of persistent instances of the **Department** class and associate some employees with them.

The **main** function of the **Top** class might contain the following:

```
Transaction t = Transaction.begin(ObjectStore.UPDATE);

//insert your code here
//Creating a Department instance
Department dep1 = new Department();
dep1.departmentName = "Marketing";
dep1.employees = new Employee[10];

//Adding dep1 to the extent of Departments
dep1.updateExtents(db, true);

t.commit();
```

This code creates a new **Department** instance and sets its name to **Marketing**.

The **updateExtent** call inserts the newly created **Department** into the proper extent. Given that the class **extent** is allocated persistently (it is reachable from a root), the **dep1** object instance automatically becomes persistent.

For more information

To learn more about roots, entry-point objects, and collections, see the *ObjectStoreJava API User Guide*.

Now add some **Employees** and attach them to **Marketing**:

```
//Creating some Employee instances
Employee e1= new Employee(10, 20000, "John", 33);
Employee e2= new Employee(25, 35000, "Peter", 40);

//Attaching the employees to the Department
dep1.employees[0]= e1;
dep1.employees[1]= e2;
```

When you attach **e1** and **e2** to the **employees** array, they become persistent by reachability because **dep1** is a persistent object.

To complete the example, add another **Department** with some additional **employees**.

```
//Creating another Department instance
Department dep2 = new Department();
dep2.departmentName = "Sales";
dep2.employees = new Employee[10];

//Adding dep2 to the extent of Departments
dep2.updateExtents(db, true);

//Creating some Employee instances
Employee e3= new Employee(10, 50000, "Chris", 51);
Employee e4= new Employee(45, 38000, "Larry", 39);
Employee e5= new Employee(15, 35000, "Andy", 55);

//Attaching the employees to the Department
dep2.employees[0]= e3;
dep2.employees[1]= e4;
dep2.employees[2]= e5;
```

Building the Project

This section shows you how to build a project, using either the Java JDK or Visual J++.

Building the Project with JDK

The Component Wizard automatically generates a batch file called **doall.bat** that contains the commands you need to compile and execute your project.

The **doall.bat** file

The following is the contents of the **doall.bat** file:

```
@echo off

ECHO Compiling source files
javac *.java

ECHO Running ObjectStore classfile postprocessor
call osjcfp @cfpargs

ECHO run the program
java Personnel.Top
```

The **doall.bat** assumes that you are using the **javac** command-line compiler and that it is in your **bin** path. If this is not the case, make the appropriate changes to **doall.bat**.

After all the Java files are compiled, **doall.bat** calls the postprocessor to process all the classes that are part of the schema you defined with the Database Designer.

How to build a project

To build a project with JDK:

1 Make the directory where you generated your project your current directory.

2 Run **doall.bat**.

The **cfpargs** file

The **cfpargs** file is generated by the Component Wizard and contains the flags for the postprocessor. This is the contents of the **cfpargs** file for the **personnel** example:

```
-dest . -inplace
-persistaware
Top.class
Extents.class
Ext_OSTreeSet.class
Ext_OSVector.class
-persistcapable
Customer.class
Department.class
Employee.class
Person.class
```

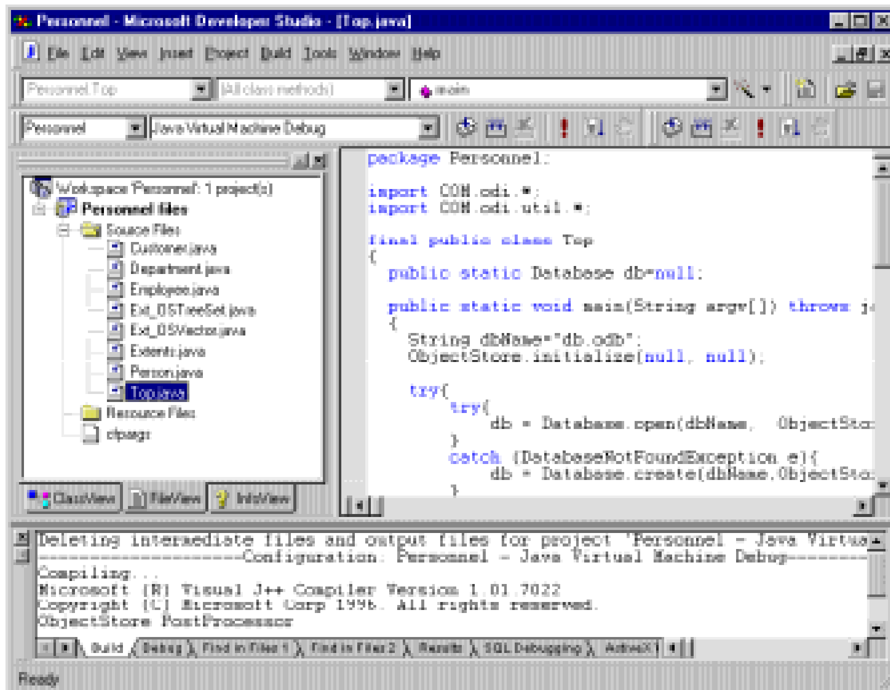
If the compilation and postprocessing operations successfully complete, the **java Personnel.TOP** command executes your application.

For more information

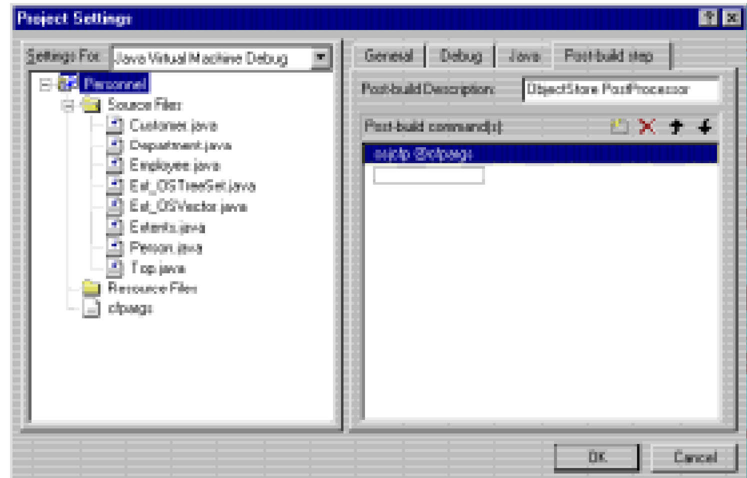
See the postprocessor documentation in the *ObjectStoreJava API User Guide.*

Building the Project Using Visual J++ 1.1

When you run the Component Wizard from Visual J++, it generates the VJ++ package that includes all the Java files needed to compile the application based on the schema you defined with the Database Designer.



The Component Wizard also sets the project so that at the end of the compilation process, the compiler calls the postprocessor.



To build a project

To build the project using Visual J++, select Rebuild All from the Build menu.

The following code is a sample of the output of the compilation process:

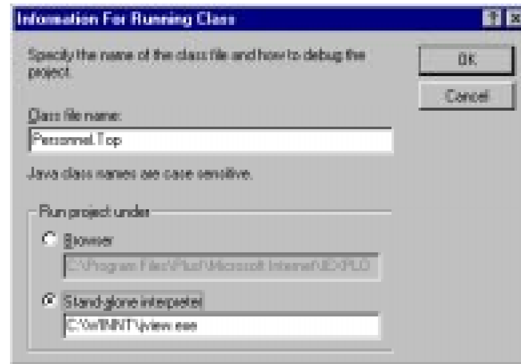
```
Deleting intermediate files and output files for project 'Personnel - Java Virtual Ma
--------------------Configuration: Personnel - Java Virtual Machine Debug------------
Compiling...
Microsoft (R) Visual J++ Compiler Version 1.01.7022
Copyright (C) Microsoft Corp 1996. All rights reserved.
ObjectStore PostProcessor

Personnel - 0 error(s), 0 warning(s)
```

To run the application

To run the application, select Execute from the Build menu, or press Ctrl+F5.

The first time you run the application, Visual J++ prompts you to enter the name of the class you want to execute. In this example, you enter **Personnel.Top**.



If Visual J++ displays “ERROR: Could not execute Personnel.Top: The system cannot find the file specified”, check that you added the proper class path to the list of class directories. For example, if you created your project under **c:\Myprojects\Personnel**, make sure the **c:\Myprojects** directory is included in the list of class directories:

To add a directory to the Class path:

1 Select Options from the Tools menu.

2 Click the Directories tab.

3 Select Class File from the Show directories for: list and then click OK.