IBM ILOG OPL V6.3

IBM ILOG OPL Interface User's Manual

Copyright

Trademarks

Acknowledgement

The language manuals are based on, and include substantial material from, The OPL Optimization Programming Language by Pascal Van Hentenryck, © 1999 Massachusetts Institute of Technology.

Table of contents

Interfaces User's Manual

IBM® ILOG® OPL Interfaces enable users to integrate OPL models with IBM ILOG Concert. They are available in the C++, Java, and .NET programming languages. This manual provides a tutorial approach to using these application programming interfaces (APIs), and should be used in conjunction with the *Interfaces Reference Manual* for the language you are using.

In this section

Introduction
Explains how to use the C++, Java, and .NET libraries to integrate OPL models with IBM® ILOG® Concert Technology. The introduction to the C++, Java, and .NET interfaces stresses the specificities for each language.

Tutorial
Shows how to write basic code to create a simple OPL model from a model definition file and a model data file. The model is solved using CPLEX through the Concert API. Each step is illustrated by a code sample in each language.

Working with OPL interfaces
Explains in more general terms how to work with the OPL C++, Java and .NET Interfaces. This section is also illustrated by code samples for each language.

Introduction

Explains how to use the C++, Java, and .NET libraries to integrate OPL models with IBM® ILOG® Concert Technology. The introduction to the C++, Java, and .NET interfaces stresses the specificities for each language.

In this section

About the Interfaces User's Manual
Provides a short overview of the purpose of the OPL interfaces, the design principles underlying them, and material you should read before using this manual.

Using the C++ interface
Explains how to compile and build an application that uses OPL C++ interfaces. For details, see the *C++ API Reference Manual*.

The Java interface
Presents the Java API delivered with IBM® ILOG® OPL. For details, see the *Java API Reference Manual*.

The .NET interface
Presents the .NET API delivered with IBM® ILOG® OPL. For details, see the *.NET API Reference Manual*.

About the Interfaces User's Manual

IBM® ILOG® OPL Interfaces enable users to integrate OPL models with IBM ILOG Concert.

These interfaces are available in the C++, Java, and .NET programming languages. Each of these application programming interfaces (APIs) is documented in the *Interfaces Reference Manual* for that language, accessible from the table of contents.

Before you start

Before reading this manual, we recommend that you read the part about Languages and APIs in the *CPLEX User's Manual*, which presents IBM ILOG Concert for the various programming languages. IBM ILOG Concert is used by both solving engines, CPLEX® and CP Optimizer.

Also make sure you read How to read the OPL documentation for details of prerequisites, conventions, documentation formats, and other general information.

When to use the API

The recommended approach to modeling and solving a problem is to start with the IBM ILOG OPL IDE.

Later, you may wish to use an API to extend your model.

Design principles

The design principles for the OPL interfaces are:

- separate the model from the solver
- instantiate the same OPL model with different data
- allow Concert to access and modify the model
- provide data easily
- access data and results

Using the C++ interface

The C++ API of OPL is based on the C++ Concert Technology, on the CPLEX® C++ API, and on the CP Optimizer API which are themselves based on Concert Technology. For more information, see the *Concert Reference Manual*.

Important: **Microsoft limitation:** The C++ OPL API (VS 2005 and VS 2008) example projects contain a post-build event specified as

```
mt.exe /nologo -outputresource:$(TargetFileName);1 -manifest
examples.manifest
```

This event is required for C++ applications compiled with Visual Studio 2005 and 2008. By default, all applications built with these versions of Visual Studio are built as isolated applications with a manifest, either embedded as a resource, or accompanying the final binary as an external file. Sometimes the manifest generated automatically is not correct. In this case, you need to force generation of the right manifest.

To compile and build an application that uses OPL C++ interfaces:

1. Include the directory of header files:

   `<OPL_dir>\include`

2. Link with the following IBM ILOG libraries:

Windows libraries	Unix libraries except AIX	AIX libraries
concert.lib	concert.so	concert.a
cp.lib	cp.so	cp.a
cplex <*version_number*> .lib	cplex <*version_number*> .so	cplex <*version_number*> .a
dbkernel.lib	dbkernel.so	dbkernel.a
dblnkdyn.lib	dblnkdyn.so	dblnkdyn.a
iljs.lib	iljs.so	iljs.a
ilocplex.lib	ilocplex.so	ilocplex.a
ilog.lib	ilog.so	ilog.a
opl.lib	opl.so	opl.a

   These libraries are in the following directory:

   `<OPL_dir>\lib\<port_name>\<format>`

   where `<OPL_dir>` is your installation directory.

3. Make sure the following directory is in your `PATH` (on Windows), `LIBPATH` (AIX platforms), or `LD_LIBRARY_PATH` (other Unix platforms) environment variable:

   `<OPL_dir>\bin\<port_name>\<format>`

   It contains the shared libraries (Windows `.dll` and Unix `.so` or `.a` files) that applications need to run. When you install OPL on Windows, this directory is added to the `PATH`, `LIBPATH`, or `LD_LIBRARY_PATH` variable.

4. Set your environment variables.

   - **On Windows**, make sure that the path

     `<OPL_dir>\bin\<port_name>`

     is in your PATH environment variable, so that the shared libraries are found.

   - **On Unix**, make sure that

     `<OPL_dir>\bin\<port_name>`

     is in your `LIBPATH` or `LD_LIBRARY_PATH` environment variable, so that the shared libraries are found.

Note: To see the solution in your file, you have to force the flush of the output stream, using

```
cout<<endl;
```

For more information and examples on how to use Concert Technology, as well as the CPLEX® and CP Optimizer APIs for Concert Technology, see the Concert, CPLEX and CP Optimizer documentation.

The Java interface

Presents the Java API delivered with IBM® ILOG® OPL. For details, see the *Java API Reference Manual*.

In this section

Overview of the Java interface
Presents a high-level overview of the OPL Java interface.

Object creation and factories
Describes the usage of constructors in the OPL Java interface.

Memory management
Provides information on memory management while using the OPL Java interface.

Compatibility with Java CPLEX
Describes the relationship of the `ilog.concert` and `ilog.cplex` packages with their equivalent packages in IBM® ILOG® CPLEX® .

Compatibility with CP Optimizer Java interface
The Java interface is fully compatible with the CP Optimizer Java interface

Deployment of Java applications
Provides information on how to deploy an application that uses the OPL Java interface.

Overview of the Java interface

The Java interface offers basically the same functionality as the C++ one.

The Java API is split into several packages:

- The `ilog.concert` package contains the Concert modeling API, for modifying models.
- The `ilog.cp` package contains the CP Optimizer control API, for controlling the solving process of constraint programming models.
- The `ilog.cplex` package contains the CPLEX® control API, for controlling the solving process of mathematical programming models.
- The `ilog.opl` package contains the OPL control API, for loading and accessing models.

The Java API is written as a JNI wrapper on the equivalent C++ libraries; it offers the same functionality as the C++ API.

Each call to a method of the API goes through a wrapping layer. This may result in a slight performance overhead while the model is created, compared to using the C++ API, depending on the number of API function calls. Since you call only few API functions to load and solve your model, the overhead is negligible in usual cases, but it may become important if you use the low-level Concert, CP Optimizer, or CPLEX API for a complete model creation (for example, constructing a matrix line by line using `IloNumExpr` APIs or adding `IloConstraint` objects one by one to an `IloModel` using the API). It is therefore recommended to use the OPL language to model your problems whenever possible, and use only the low-level Concert APIs for the parts that need it (runtime additions, etc.).

Once created, the model is still solved fully in C++, so there is no loss of performance when solving models, whatever language you choose.

Object creation and factories

The C++ API allows you to create objects using a constructor. To provide greater flexibility and allow for evolution, the Java API uses a “Factory” pattern: objects are created by calls to the methods of a root object. For example, you create OPL objects using the methods of the class `IloOPLFactory.createOPLModel`, `IloOPLFactory.createCPLEX`, and so on. Similarly, you create Concert modeling constructs using the methods of the class `IloMPModeler` (implemented by the `IloCplex` class): `numVar`, `range`, `minimize`, and so on.

For example, the following code lines create an instance of `IloNumVar`:

```
IloMPModeler modeler=new IloCplex();
IloNumVar var=modeler.numVar(0,10);
```

The Concert C++ modeling API works by redefining operators to provide a compact notation for common constructs. In Java, the equivalent constructs are created through regular methods of `IloModeler`: `ge`, `eq`, `prod`, and so on.

Memory management

The Java garbage collector usually takes care of the memory allocated by Java objects. However, since the OPL Java API allocates memory in C++ as well, the memory management is slightly different. When you use the OPL Java API, the first object you create is always the OPL factory. Then, you use the OPL factory to create all other objects. All the C++ memory is allocated on an internal heap of the `IloOplFactory` object and cleaned up by a call to the method `IloOPLFactory.end`. The internal heap is an instance of the C++ class `IloEnv`.

Important: This means that in most cases you do not need to be concerned with memory management: all the memory used by your model is correctly cleaned up at the end.

Some applications may need tighter control on memory management.

This is the case for applications to which **all** of the following applies:

- They demand a lot of memory.

  and

- They make a lot of incremental model modifications: elements are repeatedly added to, then removed from, the model, which is solved after each addition or removal.

  and

- They are long-lived, that is, the application keeps modifying and re-solving the same model over long periods.

Such applications can explicitly manage memory by calling the method `IloMPModeler.delete` on Concert objects or `end` on OPL objects. These methods delete objects before the global cleanup and thus free memory earlier.

Compatibility with Java CPLEX

The `ilog.concert` and `ilog.cplex` packages are designed to be compatible with the equivalent packages in IBM® ILOG® CPLEX® . This means that they offer the same API (slightly extended for OPL), although the implementation is quite different. The benefit is that you can reuse your existing JConcert/Java CPLEX modeling code and combine it with OPL models.

Because OPL can produce full Concert models and offers a backward compatible API, you benefit from a smooth migration path to OPL without losing your previous work.

Note:

1. The CPLEX Java API related to `IloLPMatrix`, which allows direct manipulation of the CPLEX matrix in Java, is not available in the OPL implementation.
2. Serialization is not supported for JConcert modeling classes.
3. CPLEX callbacks are supported with parallel search in Java interfaces.

Compatibility with CP Optimizer Java interface

The Java interface is fully compatible with the CP Optimizer Java interface. The `ilog.cp` package shipped with OPL is identical to that shipped with CP Optimizer.

Deployment of Java applications

To compile and build an application that uses OPL Java interfaces, you need only one JAR file: `oplall.jar`, located in

`<OPL_dir>\lib`

This JAR file uses the dynamic library `bin\<port_name>\opl<version>.dll` (on Windows) or `bin\<port_name>\opl<version>.so` (on UNIX) at run time. The OPL Java API supports the JDK from version 5.0, on Windows and UNIX.

Make sure your `CLASSPATH` variable includes

```
<OPL_dir>\lib\oplall.jar
```

See also AIX platforms in Working Environment for limitations.

The .NET interface

Presents the .NET API delivered with IBM® ILOG® OPL. For details, see the *.NET API Reference Manual*.

In this section

Overview of the .NET interface
Presents a high-level overview of the OPL .NET interface.

Object creation and factories
Describes the usage of constructors in the OPL .NET interface.

Memory management
Provides information on memory management while using the OPL .NET interface.

CPLEX goals
Describes how the .NET API handles search strategies on CPLEX goals.

Compatibility with .NET CPLEX
Describes the relationship of the `ILOG.Concert` and `ILOG.CPLEX` namespaces with their equivalent packages in ILOG CPLEX.

Compatibility with CP Optimizer .NET interface
The .NET interface is fully compatible with the CP Optimizer .NET interface

Deployment of .NET applications (Windows only)
Provides information on how to deploy an application that uses the OPL . NET interface.

Overview of the .NET interface

The .NET interface offers basically the same functionality as the C++ one.

The .NET API is split into several namespaces:

- The `ILOG.Concert` namespace contains the Concert modeling API, for modifying models.
- The `ILOG.CP` namespace contains the CP Optimizer control API, for controlling the solving process of constraint programming models.
- The `ILOG.CPLEX` namespace contains the CPLEX control API, for controlling the solving process of mathematical programming models.
- The `ILOG.OPL` namespace contains the OPL control API, for loading and accessing models.

This API is available for all the languages supported by the .NET platform. Examples are provided with OPL for C#, and Visual Basic.

The .NET API is written as a JNI wrapper on the equivalent C++ libraries; it offers the same functionality as the C++ API.

Each call to a method of the API goes through a wrapping layer. This may result in a slight performance overhead while the model is created, compared to using the C++ API, depending on the number of API function calls. Since you call only few API functions to load and solve your model, the overhead is negligible in usual cases, but it may become important if you use the low-level Concert, CP Optimizer, or CPLEX® API for a complete model creation (for example, constructing a matrix line by line using `INumExpr` APIs or adding `IConstraint` objects one by one to an `IModel` using the API). It is therefore recommended to use the OPL language to model your problems whenever possible, and use only the low-level Concert APIs for the parts that need it (runtime additions, etc.).

Note: The .NET API requires the `vjslib.dll` library.

Object creation and factories

The C++ API allows you to create objects using a constructor. To provide greater flexibility and allow for evolution, the .NET API uses a “Factory” pattern: objects are created by calls to the methods of a root object. For example, you create OPL objects using the methods of `OPLFactory.CreateOPLModel`, `OPLFactory.CreateCPLEX`, and so on. Similarly, you create Concert modeling constructs using the methods of `IMPModeler` (implemented by the `Cplex` class): `NumVar`, `Range`, `Minimize`, and so on.

The Concert C++ modeling API works by redefining operators to provide a compact notation for common constructs. In .NET, the equivalent constructs are created through regular methods of `IModeler`: `Ge`, `Eq`, `Prod`, and so on.

Memory management

The .NET CLR garbage collector usually takes care of the memory allocated by .NET objects. However, since the OPL .NET API allocates memory in C++ as well, the memory management is slightly different.

When you use the OPL .NET API, the first object you create is always the OPL factory. Then, you use the OPL factory to create all other objects. All the C++ memory is allocated on an internal heap of the `OplFactory` object and cleaned up by a call to the method `OPLFactory.End`. The internal heap is an instance of the C++ class `IloEnv`.

Important: This means that in most cases you do not need to be concerned with memory management: all the memory used by your model is correctly cleaned up at the end.

Some applications may need tighter control on memory management.

This is the case for applications to which all of the following apply:

- They demand a lot of memory.

  and

- They make a lot of incremental model modifications: elements are repeatedly added to, then removed from, the model, which is solved after each addition and removal.

  and

- They are long-lived, that is, the application keeps modifying and re-solving the same model over long periods.

Such applications can explicitly manage memory by calling the method `IMPModeler.Delete` on Concert objects or `End` on OPL objects. These methods delete objects before the global cleanup and thus free memory earlier.

CPLEX goals

IBM® ILOG® CPLEX® provides advanced control on search strategies through user goals that are called during the search. This feature is not available for the .NET API in this release (although it is supported in C++ and Java). It will be supported in a future release. However, CPLEX callbacks, including CPLEX callbacks with parallel search, **are** supported in this release of OPL.

Compatibility with .NET CPLEX

The `ILOG.Concert` and `ILOG.CPLEX` namespaces are designed to be compatible with the equivalent namespaces in IBM® ILOG® CPLEX® . This means that they offer the same API (slightly extended for OPL) although the implementation is quite different. The benefit is that you can reuse your existing Concert.NET or CPLEX.NET modeling code and combine it with OPL models. Because OPL can produce full Concert models and offers backward-compatible APIs, you benefit from a smooth migration path to OPL without losing your previous work.

Note: The CPLEX .NET API related to `ILPMatrix`, which allows direct manipulation of the CPLEX matrix in .NET, is not available in the OPL implementation.

Compatibility with CP Optimizer .NET interface

The .NET interface is fully compatible with the CP Optimizer .NET interface. The IBM® ILOG CP .NET API shipped with OPL is identical to that shipped with CP Optimizer.

Deployment of .NET applications (Windows only)

The .NET API is provided as the assembly file `lib/oplall.dll`, which uses the dynamic library `bin/opl<version_number>_dotnet.dll` at run time. The OPL .NET API supports version 2.0 of .NET Framework.

More specifically, to compile and build an application that uses OPL .NET interfaces, you need:

- one DLL file: `oplall.dll`, located in `OPL_dir\lib`
- Microsoft .NET Framework Version 2.0 Redistributable Package. This package is included in Microsoft Visual Studio. You can also download it for free from

  http://www.msdn.microsoft.com/netframework/

  The appropriate version of Visual Studio is VS2005 for .NET2.0.

Tutorial

Shows how to write basic code to create a simple OPL model from a model definition file and a model data file. The model is solved using CPLEX through the Concert API. Each step is illustrated by a code sample in each language.

In this section

Overview of the Tutorial
Reviews the design principles of the OPL APIs, where to locate the libraries and their respective code samples in the distribution, and presents the different sections of this tutorial.

Creating an OPL model
Shows how to create an OPL model that utilizes the OPL interfaces.

Specifying a data source
Shows how to specify the data source for your OPL model.

Generating the Concert model
Describes how to generate the Concert model for your OPL model.

Solving the model
Shows how to solve your model.

Accessing the solution
Provides information on how to access the solution through OPL and through Concert.

Using run configuration and projects
Shows how to use run configurations to create the project and access the run configuration and the model using Concert.

Overview of the Tutorial

The IBM® ILOG® OPL Interfaces library enables users to integrate OPL modeling with IBM ILOG Concert Technology.

These interfaces are designed to fulfill several goals, chiefly:

- separate the model from the data
- separate the modeling phase from the solving phase
- instantiate the same OPL model definition with different data
- embed with IBM ILOG Concert Technology
- provide custom data easily

OPL Interfaces are available in C++, Java, and .NET. See *Introduction* for a general presentation of the Interfaces in the three languages. See also the *Interfaces Reference Manuals* in

`<OPL_dir>\doc\html\en-US\refcppopl\index.html`

`<OPL_dir>\doc\html\en-US\refjavaopl\index.html`

`<OPL_dir>\doc\chm\index.chm`

where `<OPL_dir>` is your installation directory.

Most of the code snippets used in this tutorial are extracted from the `mulprod` example, which exists in all four languages at the following locations:

<OPL_dir>\examples\opl_interfaces\cpp\src\mulprod.cpp

<OPL_dir>\examples\opl_interfaces\java\mulprod\src\mulprod\Mulprod.java

<OPL_dir>\examples\opl_interfaces\dotnet\x86_.net2005_8.0\VisualBasic\Mulprod\Mulprod.vb

<OPL_dir>\examples\opl_interfaces\dotnet\x86_.net2005_8.0\CSharp\Mullprod\Mulprod.cs

A general presentation of the code samples is provided in Interfaces examples in the *Language and Interfaces Samples* manual.

In this tutorial, the .NET code samples from `Mulprod.vb` are written in Visual Basic. The instructions that cannot be illustrated in each language are given by default in C++. This tutorial walks you through the procedures.

Creating an OPL model

To create an OPL model using the IBM® ILOG® OPL Interfaces library, you need:

1. *To create the Concert environment*.
2. *To create the error handler* in the environment.
3. *To identify the model source*.
4. *To identify the model definition*.
5. *To create the engine instance*.
6. *To create the OPL model*.

These steps are explained in this topic.

To create the Concert environment

As for any IBM® ILOG® Concert Technology model, you need an instance of the environment in which to create your model objects.

- Write the following code.

  C++
  ```
  IloEnv env;
  ```

  Java
  ```
  IloOplFactory oplF = new IloOplFactory();
  ```

  .NET (Visual Basic)
  ```
  Dim oplF As OplFactory = New OplFactory
  ```

To create the error handler

- Create an error handler is necessary in the environment to report errors and warnings during the translation of the model text.

  C++
  ```
  IloOplErrorHandler handler(env,cout);
  ```

  Java
  ```
  IloOplErrorHandler errHandler = oplF.createOplErrorHandler();
  ```

  .NET (Visual Basic)
  ```
  Dim errorHandler As OplErrorHandler = oplF.CreateOplErrorHandler()
  ```

To identify the model source

- Pass the model source that provides the text to interpret.

C++

```
      IloOplModelSource modelSource(env, DATADIR 'mulprod.mod');
```

Java

```
      IloOplModelSource modelSource = oplF.createOplModelSource(DATADIR
+ '/mulprod.mod');
```

.NET (Visual Basic)

```
     Dim modelSource As OplModelSource = oplF.CreateOplModelSource(DATADIR
 + '/mulprod.mod')
```

To identify the model definition

- Use same model definition to instantiate one or more models.

C++

```
      IloOplSettings settings(env,handler);
      IloOplModelDefinition def(modelSource,settings);
```

Java

```
      IloOplModelDefinition def = oplF.createOplModelDefinition
(modelSource,settings);
```

.NET (Visual Basic)

```
      Dim def As OplModelDefinition = oplF.CreateOplModelDefinition
(modelSource, settings)
```

To create the engine instance

- Create the instance of the algorithm to use for this model.

C++

```
   IloCplex cplex(env);
```

Note: If the model is to be solved by CP Optimizer engine, you would instantiate an `IloCP` object using

```
IloCP cp(env)
```

Java

```
   IloCplex cplex = oplF.createCplex();
```

Note: If the model is to be solved by CP Optimizer engine, you would instantiate an `IloCP` object using

```
IloCP cp = oplF.createCP()
```

.NET (Visual Basic)

```
Dim cplex As Cplex = oplF.CreateCplex()
```

Note: If the model is to be solved by CP Optimizer engine, you would instantiate an `IloCP` object using

```
Dim cp As CP = oplF.CreateCP()
```

To create the OPL model

- You can now create the OPL model. The constructor takes a model definition instance and an instance of `IloCplex` .

C++

```
IloOplModel opl(def,cplex);
```

Java

```
IloOplModel opl = oplF.createOplModel(def, cplex);
```

.NET (Visual Basic)

```
Dim opl As OplModel = oplF.CreateOplModel(def, cplex)
```

Specifying a data source

- In order to generate the Concert model, you need to provide data, just as you would add a data file in an OPL project along with the model file. The simplest way to get data is also to provide a file.

C++

```
        IloOplDataSource dataSource(env, DATADIR "mulprod.dat");
        opl.addDataSource(dataSource);
```

Java

```
        IloOplDataSource dataSource = oplF.createOplDataSource(DATADIR
                + "/mulprod.dat");
        opl.addDataSource(dataSource);
```

.NET (Visual Basic)

```
${snippet_N32324}
```

Generating the Concert model

- Once you have specified your data source, you can generate the Concert model.

  C++

  ```
          opl.generate();
  ```

  Java

  ```
          opl.generate();
  ```

  .NET (Visual Basic)

  ```
  ${snippet_N32380}
  ```

This method also loads the model into the engine instance passed earlier.

Solving the model

- You can solve the model in the usual way with Concert Technology.

C++

```
        if ( cplex.solve() ) {
```

Java

```
        if (cplex.solve())
```

.NET (Visual Basic)

```
${snippet_N323F2}
```

Accessing the solution

You can access the solution directly through OPL or through the Concert API.

Accessing the solution through OPL

- Print the OPL solution directly like this:

  C++

  ```
  opl.printSolution(cout);
  ```

  Java

  ```
  opl.printSolution(System.out);
  ```

  .NET (Visual Basic)

  ```
  ${snippet_N3247F}
  ```

Accessing the solution through Concert

- You can use the typical Concert API to access results.

  C++

  ```
  cout << endl
       << "OBJECTIVE: " << fixed << setprecision(2) << opl.
  getCplex().getObjValue()
       << endl;
  ```

  Java

  ```
  System.out.println("OBJECTIVE: " + opl.getCplex().getObjValue
  ());
  ```

  .NET (Visual Basic)

  ```
  ${snippet_N324EE}
  ```

In the same way, you can ask the `IloCplex` instance for the values of the variables from the OPL model. See *Using OPL model instances* for details.

Using run configuration and projects

Sometimes, it is not necessary to create intermediate objects for the model definition or the data sources as explained in *Creating an OPL model*. This is the case, for example, when you do not plan to use the data source object for various different OPL models. You can then use the classes `IloOplProject` and `IloOplRunConfiguration` to create the `IloOplModel` instance directly.

This section demonstrates this feature using the `oplrun` example which exists in all four languages at the following locations:

<OPL_dir>\examples\opl_interfaces\cpp\src\oplrunsample.cpp

<OPL_dir>\examples\opl_interfaces\java\oplrunsample\src\oplrunsample\OplRunSample.java

<OPL_dir>\examples\opl_interfaces\dotnet\x86_net2005_8.0\VisualBasic\OplRunSample\OplRunSample.vb

<OPL_dir>\examples\opl_interfaces\dotnet\x86_net2005_8.0\CSharp\OplRunSample\OplRunSample.cs

where `<OPL_dir>` is your installation directory.

Creating the project

- You can create an `IloOplProject` instance directly using a project path.

  C++

  ```
  IloOplProject prj(_env,_cl.getProjectPath());
  ```

  Java

  ```
  IloOplProject prj = oplF.createOplProject(_cl.getProjectPath());
  ```

  C#

  ```
  Dim prj As OplProject = oplF.CreateOplProject(_cl.ProjectPath)
  ```

Accessing a run configuration

- From that project, you can access one of the included run configuration. If you pass no argument, you get the default run configuration.

  C++

  ```
  rc = prj.makeRunConfiguration(_cl.getRunConfigurationName());
  ```

  Java

  ```
  rc = prj.makeRunConfiguration(_cl.getRunConfigurationName());
  ```

  C#

  ```
  rc = prj.MakeRunConfiguration(_cl.RunConfigurationName)
  ```

OPL creates the `IloOplModel` instance from the run configuration automatically.

Accessing the model

- You can access with the following code.

  C++

  ```
  IloOplModel opl = rc.getOplModel();
  ```

  Java

  ```
  IloOplModel opl = rc.getOplModel();
  ```

  C#

  ```
  Dim opl As OplModel = rc.GetOplModel()
  ```

You can then use the `IloOplModel` instance as usual.

Using the `IloOplProject` and `IloOplRunConfiguration` API brings more than one benefit:

- It is easier to use when intermediate structures are not necessary.
- It enables you to use settings files (`.ops`).
- All the advantages of run configurations remain available. In particular, it is possible to easily run the same model with different setting files or with different data sets.

Working with OPL interfaces

Explains in more general terms how to work with the OPL C++, Java and .NET Interfaces. This section is also illustrated by code samples for each language.

In this section

Using OPL model instances
Explains how to instantiate OPL models using a model definition, data, and a solving engine.

Services on solutions
Mentions additional services available through the `IloOplModel` instance as soon as a solution is available.

Custom data sources
Describes how to provide custom data sources for your model by extending the class `IloOplDataSourceBaseI`.

Accessing elements
Describes what APIs to use to access model elements such as decision variables and their values, and to iterate through model elements.

Settings
Describes the various Settings options available to customize the behavior of OPL.

Postprocessing solutions
Shows how to initiate the call to postprocessing in your model.

Error handling
Describes how to handle messages and integrate them with your environment.

Debug mode
Explains how to work with your model in debug mode.

Printing data to a stream
Shows how to print your data to a stream, using the `.dat` file syntax.

Integrating OPL with Excel using Visual Studio Tools for Office
Explains how to create a Microsoft Excel Add-in. It is coded in C# and uses OPL to solve the warehouse problem within Excel 2003.

Memory management
Provides recommendations to manage memory in Concert applications.

Using OPL model instances

Explains how to instantiate OPL models using a model definition, data, and a solving engine.

In this section

Overview
Presents basic information about working with OPL model instances.

The model definition
Shows how to specify a model definition that can then be used to instantiate one or more models.

The data
Shows how to specify the data source for your model.

Overview

To instantiate an OPL model, you need its definition and data, as well as an engine that will solve the model. Most of the code samples used in this topic are extracted for each language from the `warehouse` example, which exists in all four languages at the following locations:

`<OPL_dir>\examples\opl_interfaces\cpp\src\warehouse.cpp`

`<OPL_dir>\examples\opl_interfaces\java\warehouse\src\warehouse\Warehouse.java`

`<OPL_dir>\examples\opl_interfaces\dotnet\x86_.net2005_8.0\VisualBasic\Warehouse\Warehouse.vb`

`<OPL_dir>\examples\opl_interfaces\dotnet\x86_.net2005_8.0\CSharp\Warehouse\Warehouse.cs`

where `<OPL_dir>` is your installation directory.

A general presentation of the code samples is provided in Interfaces examples in the *Language and Interfaces Samples* manual.

In this chapter, the .NET code samples from `Warehouse.cs` are written in C#. The instructions that cannot be illustrated in each language are given by default in C++.

The model definition

The model source provides the text to interpret. An error handler is necessary to report errors and warnings during the translation of the model text. Later, you can use the same model definition to instantiate one or more models.

C++

C++: Specifying the model definition

```
int main(int argc,char* argv[]) {
        IloEnv env;


    int status = 127;
    try {
        IloCplex cplex(env);
        IloOplErrorHandler handler(env,cout);
        std::istringstream in( getModelText() );
        IloOplModelSource modelSource(env,in,"warehouse");
        IloOplSettings settings(env,handler);
        IloOplModelDefinition def(modelSource,settings);
        IloOplModel opl(def,cplex);
```

Java

Java: Specifying the model definition

```
        IloOplFactory.setDebugMode(true);
        IloOplFactory oplF = new IloOplFactory();
       IloOplErrorHandler errHandler = oplF.createOplErrorHandler(System.out)
;
        IloCplex cplex = oplF.createCplex();
        IloOplModelSource modelSource=oplF.createOplModelSourceFromString
(getModelText(),"warehouse");
        IloOplSettings settings = oplF.createOplSettings(errHandler);
        IloOplModelDefinition def=oplF.createOplModelDefinition
(modelSource,settings);
        IloOplModel opl=oplF.createOplModel(def,cplex);
```

.NET (C#)

.NET: Specifying the model definition

```
                OplFactory.DebugMode = true;
                OplFactory oplF = new OplFactory();
              OplErrorHandler errHandler = oplF.CreateOplErrorHandler(Console.
Out);
                Cplex cplex = oplF.CreateCplex();
              OplModelSource modelSource = oplF.CreateOplModelSourceFromString
(GetModelText(), "warehouse");
                OplSettings settings = oplF.CreateOplSettings(errHandler);
```

```
                OplModelDefinition def = oplF.CreateOplModelDefinition
(modelSource, settings);
                OplModel opl = oplF.CreateOplModel(def, cplex);
```

The data

To generate the CPLEX® model, OPL needs to know where to take the data from. You must therefore specify a data source.

The data source can be:

- either an OPL data file (as in the *Tutorial*) which allows access to data in files or databases, or
- a custom-coded data source, as in the examples below. (The custom data source is described in *Custom data sources*.)

C++

C++: Specifying a data source

Java

Java: Specifying a data source

```
        IloOplDataSource dataSource=new MyParams
(oplF,nbWarehouses,nbStores,fixed,disaggregate);
        opl.addDataSource(dataSource);
        opl.generate();
```

.NET (C#)

.NET: Specifying a data source

```
                OplDataSource dataSource = new MyParams(oplF, nbWarehouses,
nbStores, fixedP, disaggregate);
                opl.AddDataSource(dataSource);
                opl.Generate();
```

Now, the OPL model is available from CPLEX® . You can use the full `IloCplex` API to solve the model. You can reuse the same `cplex` instance for different OPL models. However, when you access for the first time a postprocessing model element that uses variable values in its definition, you may get an “unbound variable” exception if the `cplex` instance has been used later for another model and hence is not synchronized with the invoking model anymore.

Services on solutions

Additional services are available through the `IloOplModel` instance as soon as a solution is available. The values for decision variables within the solution are accessible by their names. See *Accessing elements*.

Custom data sources

You can provide OPL models with custom data sources by extending the class `IloOplDataSourceBaseI` . All you need to do is implement the `read` method, which will be called by the OPL interpreter as necessary. A custom data source uses the interface `IloOplDataHandler` to transfer data to the model, and the method `getDataHandler` to access the associated data handler and send events. The handler API is event-driven, similar to SAX interfaces for XML.

Two examples of how these handlers could be used to initialize custom data sources are given below:

Initialization of a multi-dimensional array custom data source

```
            handler.StartElement("cost")
            handler.StartIndexedArray()
            For i = 1 To NUMDEMAND
                handler.SetItemStringIndex(dtin.Rows(i - 1)(0))
                handler.StartIndexedArray()
                For j = 1 To NUMSUPPLY
                    handler.SetItemStringIndex(dtin.Columns(j).ColumnName)
                    handler.AddNumItem(dtin.Rows(i - 1)(j))
                Next j
                handler.EndIndexedArray()
            Next i
            handler.EndIndexedArray()
            handler.EndElement()
```

Initialization of a string set custom data source

```
            handler.StartElement("Plants")
            handler.StartSet()
            For j = 1 To NUMSUPPLY
                handler.AddStringItem(dtin.Columns(j).ColumnName)
            Next j
            handler.EndSet()
            handler.EndElement()
```

C++

C++: Providing custom data sources

```
void MyParams::read() const {
    IloOplDataHandler handler = getDataHandler();

    handler.startElement("nbWarehouses");
    handler.addIntItem(_nbWarehouses);
    handler.endElement();

    handler.startElement("nbStores");
    handler.addIntItem(_nbStores);
    handler.endElement();
```

```
    handler.startElement("fixed");
    handler.addIntItem(_fixed);
    handler.endElement();

    handler.startElement("disaggregate");
    handler.addIntItem(_disaggregate);
    handler.endElement();
}
```

Java

Java: Providing custom data sources

```
    static class MyParams extends IloCustomOplDataSource
    {
        int _nbWarehouses;
        int _nbStores;
        int _fixed;
        int _disaggregate;

       MyParams(IloOplFactory oplF,int nbWarehouses,int nbStores,int fixed,int
disaggregate)
        {
            super(oplF);
            _nbWarehouses = nbWarehouses;
            _nbStores = nbStores;
            _fixed = fixed;
            _disaggregate = disaggregate;
        }

        public void customRead()
        {
            IloOplDataHandler handler = getDataHandler();

            handler.startElement("nbWarehouses");
            handler.addIntItem(_nbWarehouses);
            handler.endElement();

            handler.startElement("nbStores");
            handler.addIntItem(_nbStores);
            handler.endElement();

            handler.startElement("fixed");
            handler.addIntItem(_fixed);
            handler.endElement();

            handler.startElement("disaggregate");
            handler.addIntItem(_disaggregate);
            handler.endElement();
        }
    };
```

.NET (C#)

Accessing the values a decision variable within a solution

```
        internal class MyParams : CustomOplDataSource
        {
                int _nbWarehouses;
                int _nbStores;
                int _fixed;
                int _disaggregate;

                internal MyParams(OplFactory oplF,int nbWarehouses,int
nbStores,int fixedP,int disaggregate)
                        : base(oplF)
                {
                        _nbWarehouses = nbWarehouses;
                        _nbStores = nbStores;
                        _fixed = fixedP;
                        _disaggregate = disaggregate;
                }
```

Accessing elements

Describes what APIs to use to access model elements such as decision variables and their values, and to iterate through model elements.

In this section

Accessing model elements
Shows how to access the elements of your model.

Iterating through OPL elements
Shows how to iterate through the elements of your model.

Accessing model elements

To access any model element by its name:

- Use the method `getElement`.

  C++

  ```
  IloOplElement supply = oplModel.getElement("supply");
  ```

  Java

  ```
  IloOplElement supply = oplModel.getElement("supply");
  ```

  .NET

  ```
  OplElement supply = oplModel.GetElement("supply");
  ```

The `IloOplElement` interface offers accessors for all possible element types. It is the user's responsibility to pick the right accessor for the type of the elements he has declared.

For decision variables, there are two different types of accessors:

- one to obtain the Concert object itself for this decision variable,
- the other to get the values of decision variables within a solution. The latter is available only if a solution has been found.

The subsequent sections give examples of each for a `supply` model element.

To access a decision variable:

- To get the `supplyVar1` decision variable within a model, write the following code.

  C++

  ```
  IloNumVarMap supplyVarMap = supply.asNumVarMap();
  IloNumVar supplyVar1 = supplyVarMap.get(1);
  ```

  Java

  ```
  IloNumVarMap supplyVarMap = supply.asNumVarMap();
  IloNumVar supplyVar1 = supplyVarMap.get(1);
  ```

  .NET

  ```
  INumVarMap supplyVarMap = supply.AsNumVarMap();
  INumVar supplyVar1 = supplyVarMap.Get(1);
  ```

To access the values of a decision variable:

- To get the values of the `supply1` decision variable within a solution, write the following code.

C++

```
IloNumMap supplyMap = supply.asNumMap();
double supply1 = supplyMap.get(1);
```

Java

```
IloNumMap supplyMap = supply.asNumMap();
double supply1 = supplyMap.get(1);
```

.NET

```
INumMap supplyMap = supply.AsNumMap();
double supply1 = supplyMap.Get(1);
```

Iterating through OPL elements

The OPL Interface libraries enable your applications to iterate through OPL elements such as arrays (maps) and sets. This feature is illustrated by the `iterators` example, which contains two samples. The `iterators` example is available in C++, Java, and .NET Visual Basic and C# at the following locations:

`<OPL_dir>\examples\opl_interfaces\cpp\src\iterators.cpp`

`<OPL_dir>\examples\opl_interfaces\java\iterators\src\iterators\Iterators.java`

`<OPL_dir>\examples\dotnet\x86_.net2005_8.0\VisualBasic\Iterators\Iterators.vb`

`<OPL_dir>\examples\dotnet\x86_.net2005_8.0\CSharp\Iterators\Iterators.cs`

where `<OPL_dir>` is your installation directory.

Sample1

The purpose of `Sample1` is to check the result of filtering by iterating on the generated data element. The data element is an array of strings that is indexed by a set of strings. It is filled as the result of an iteration on a set of tuples by filtering out the duplicates. It is based on the `transp2.mod` model.

The simplified model is:

```
{string} Products = ...;
tuple Route { string p; string o; string d; }
{Route} Routes = ...;
{string} orig[p in Products] = { o | <p,o,d> in Routes };
```

Sample2

The purpose of `Sample2` is to output a multidimensional array `x[i][j]` to illustrate how arrays and subarrays are managed, as shown in *Output of a multidimensional array*.

Output of a multidimensional array

```
                IloIntMap x = opl.getElement("x").asIntMap();
                IloSymbolSet s1 = opl.getElement("s1").asSymbolSet();
                IloSymbolSet s2 = opl.getElement("s2").asSymbolSet();

                // Iterate on the first indexer.
                for (IloSymbolSetIterator it1(s1); it1.ok(); ++it1){
                        // Get the second dimension array from the first
dimension.
                        IloIntMap sub = x.getSub(*it1);
                        // Iterate on the second indexer of x (that is the
indexer of the subarray).
                        for (IloSymbolSetIterator it2(s2); it2.ok(); ++it2){
                                // This is the last dimension of the array,
so you can directly use the get method.
                                cout << *it1 << " " << *it2 << " " << sub.get
```

```
(*it2) << "\n";
                                }
                }
```

To access the elements of an array, you must first access the sub-arrays until the last dimension, then you can get the values. Here, as there are two dimensions, you have to get one sub-array from which you can directly get the values. The array of integers is indexed by two sets of strings.

The simplified model is:

```
{string} s1 = ...;
{string} s2 = ...;
{int} x[s1][s2] = ...;
```

Sample 3

The purpose of `sample3` is to output an array of tuples `arrayT[i]`, to illustrate how tuple elements can be accessed.

The simplified model is:

```
tuple t
{
int a;
int b;
}
{string} ids={"id1","id2","id3"};
t arrayT[ids]=[<1,2>,<2,3>,<1,3>];
```

Settings

There are various options available through the class `IloOplSettings` to customize the behavior of OPL.

There is an accessor to retrieve the settings of an OPL model object. You can set, for example, whether decision variable names are generated or not, or whether source locations are associated with Concert objects or not. See the C++ *Interfaces Reference Manual* for a complete list.

Postprocessing solutions

As modeling and solving are two separated phases, the OPL model does not know when a solution is available. Therefore, it does not know when to postprocess that solution. You must therefore initiate the call to postprocessing.

Note: To avoid unexpected behavior, you are recommended to call the `postprocess` method even if your model does not contain a postprocessing block.

C++

C++: Calling the postprocessing phase

Java

Java: Calling the postprocessing phase

```
        if ( cplex.solve() )
        {
            System.out.println("OBJECTIVE: " + opl.getCplex().getObjValue());

            opl.postProcess();
            opl.printSolution(System.out);
            status = 0;
        } else {
            System.out.println("No solution!");
            status = 1;
        }

        oplF.end();
```

.NET (C#)

Visualizing intermediate data

```
                if (cplex.Solve())
                {
                    Console.Out.WriteLine("OBJECTIVE: " + opl.Cplex.ObjValue)
;
                    opl.PostProcess();
                    opl.PrintSolution(Console.Out);
                    status = 0;
                }
                else
                {
                    Console.Out.WriteLine("No solution!");
                    status = 1;
                }
```

```
                oplF.End();
```

Error handling

To better integrate messages with your environment, you may choose to handle them yourself. This is possible by extending the class `IloOplErrorHandlerBaseI`.

You can override the virtual methods to report messages:

- `IloBool handleError`
- `IloBool handleWarning`
- `IloBool handleFatal`

These methods return a value to indicate whether the messages were handled correctly or not. If a message could not be handled, an exception is thrown.

It is not possible to extend the class `IloOplErrorHandler` in Java and .NET. However, you can redirect the error messages to any stream by using the appropriate factory methods:

Java

```
IloOplFactory.createOplErrorHandler(java.io.OutputStream outs)
```

.NET (C#)

```
OplFactory.CreateOplErrorHandler(TextWriter outs)
```

Debug mode

By default, the debug mode is on. It is a good practice to keep it on while you develop your application because it helps you diagnose problems. In particular, you need to have it on if you experience a core dump when running Java code. However, the debug mode slows down your application. You should therefore make sure you turn it off when you release your application. To do this, use the method `IloOplFactory.setDebugMode` .

When the default mode is on, a warning message prints to the console. The message is disabled when you turn the debug mode off. You can disable the message while still running in debug mode by a call to the method `IloOplFactory.setDebugModeWarning(false)`.

To turn off the warning, call the method

```
IloOplFactory.setDebugModeWarning(false)
```

or

```
OplFactory.DebugModeWarning = false;
```

before the `IloOplFactory` or `OplFactory` constructor is called:

```
IloOplFactory.setDebugModeWarning(false);
IloOplFactory oplF = new IloOplFactory();
IloOplErrorHandler errHandler = oplF.createOplErrorHandler();
```

The same applies if you work with .NET interfaces.

Printing data to a stream

As an additional service for testing and debugging, the class `IloOplModel` offers the possibility to print all the data to a stream, using the `.dat` file syntax.

C++

```
IloOplModel::printExternalData(ostream&)
```

Java

```
IloOplModel.printExternalData(java.io.OutputStream outs)
```

.NET (C#)

```
OplModel.PrintExternalData(TextWriter outs)
```

If your model uses data from various sources, like databases, spreadsheets, or custom sources, you will be able to write this data to one single file, which makes it easier to read. This file can be used as a data source for other model instances.

For debugging purposes, you can visualize intermediate data, that is, the data that is not provided by data sources but calculated during preprocessing.

C++

```
IloOplModel::printInternalData(ostream&)
```

Java

```
IloOplModel.printInternalData(java.io.OutputStream outs)
```

.NET (C#)

```
OplModel.PrintInternalData(TextWriter outs)
```

Integrating OPL with Excel using Visual Studio Tools for Office

Since the release of MS Office 2003, Microsoft has been working towards the integration of Office with .NET. For this purpose, Microsoft released a Visual Studio add-on named “Visual Studio Tools for Office 2003” which allows users to extend MS Office applications with any .NET language, thus bringing the benefit of the Visual Studio environment and tools to Office development.

Microsoft has replaced the VSTO 2003 add-on with **Visual Studio Tools for Office 2005** for basically the same purpose.

OPL uses this approach to integrate the OPL .NET API. This is demonstrated by the `ExcelWarehouse` example. This file is at the following location:

`<OPL_dir>\examples\opl_interfaces\dotnet\x86_.net2005_8.0\ExcelWarehouse`

where `<OPL_dir>` is your installation directory.

The solve process is triggered by a button embedded in the worksheet. The input data is taken from the worksheet and the results are written back to the worksheet.

This walkthrough explains how to create a Microsoft Excel Add-in. It is coded in C# and uses OPL to solve the warehouse problem within Excel 2003.

To create a Microsoft Excel Add-in:

1. Install MS Visual Studio 2005 and MS Office 2003.
2. Download VSTO 2005 Second Edition from this page:

   http://msdn.microsoft.com/en-us/office/aa905543.aspx

   The downloadable is free if you have VSTO 2005 or VS 2005 Professional Edition. VSTO 2005 SE allows you to create Office 2003 and 2007 add-ins from Visual Studio 2005.
3. Open Visual Studio 2005 and create a new project:
   a. Select **Visual C#/Office/2003 Add-ins** in the New Project Wizard.
   b. Name it `ExcelWarehouse`.

   A new project is created, with a dummy add-in name `ThisAddIn.cs` .

```
namespace ExcelWarehouse
     {
       public partial class ThisAddIn
          {
          private void ThisAddIn_Startup(object sender, System.
EventArgs e)
              {
                  <VSTO generated code>

              }

          private void ThisAddIn_Shutdown(object sender, System.
EventArgs e)
```

```
            {
            }

        <VSTO generated code>
        }
    }
```

4. In the `ThisAddIn.cs` file, replace the dummy code with the following lines to create a new command bar and buttons for OPL.

```
namespace ExcelWarehouse
{
   public partial class ThisAddIn
   {
     private Office.CommandBar AddInMenuBar;
     private Office.CommandBarButton SetupButton;
     private Office.CommandBarButton SolveButton;

     private void ThisAddIn_Startup(object sender, System.EventArgs e)
     {
          <VSTO generated code>

          CreateOPLCommands();
     }
     private void ThisAddIn_Shutdown(object sender, System.EventArgs e)
     {
          RemoveOPLCommands();
     }

     private void CreateOPLCommands()
     {
        try
        {
          // new command bar for OPL:
          AddInMenuBar = Application.CommandBars.Add(
            "OPL Commands", Office.MsoBarPosition.msoBarTop, missing,
true);
          // setup button:
          SetupButton = (Office.CommandBarButton)AddInMenuBar.Controls.
Add
             (Office.MsoControlType.msoControlButton, missing, missing,
                                                              missing,
true);
          SetupButton.Caption = "OPL Warehouse Setup";
          SetupButton.Style =
                          Microsoft.Office.Core.MsoButtonStyle.
msoButtonCaption;
          SetupButton.Click += new
                        Office._CommandBarButtonEvents_ClickEventHandler
(
          SetupButton_Click);

          // solve button:
```

```
          SolveButton = (Office.CommandBarButton)AddInMenuBar.Controls.
Add(
                          Office.MsoControlType.msoControlButton,
                           missing, missing, missing, true);
          SolveButton.Caption = "OPL Warehouse Solve";
          SolveButton.Style =
                          Microsoft.Office.Core.MsoButtonStyle.
msoButtonCaption;
          SolveButton.Click += new
```

```
                        Office._CommandBarButtonEvents_ClickEventHandler
(
          SolveButton_Click);
          AddInMenuBar.Visible = true;
                  }
        catch (Exception ex)
        {
          MessageBox.Show(ex.Message, ex.Source, MessageBoxButtons.OK,
                                                          MessageBoxIcon.
Error);
        }
     }

     private void RemoveOPLCommands()
     {
         SolveButton.Delete(false);
         SolveButton = null;
         SetupButton.Delete(false);
         SetupButton = null;
         AddInMenuBar = null;
     }

     private void SolveButton_Click(Office.CommandBarButton Ctrl,
                                                        ref bool
CancelDefault)
     {
       // respond to solve button click
     }

     private void SetupButton_Click(Office.CommandBarButton Ctrl, ref
bool CancelDefault)
     {
      // respond to setup button click }
     }
   }
}
```

5. Add some utility methods to get/set values in cells.

```
static private void setValue(Excel.Worksheet sheet, String cell, String
value)
{
```

```
   sheet.get_Range(cell,
      cell).set_Value(Excel.XlRangeValueDataType.xlRangeValueDefault,
value);
}

static private String getValue(Excel.Worksheet sheet, String cell)
{
   return (String)sheet.get_Range(cell, cell).Text;
}
```

6. Add the code for the **Setup** button to populate the current Excel sheet.

```
private void SetupButton_Click(Office.CommandBarButton Ctrl, ref bool

CancelDefault)
{
   Excel.Worksheet sheet =
            (Excel.Worksheet)this.Application.ActiveWorkbook.ActiveSheet;

   setValue(sheet, "A1", "Data used by the OPL model:");
   setValue(sheet, "A2", "nbWarehouses:");
   setValue(sheet, "B2", "5");
   setValue(sheet, "A3", "nbStores:");
   setValue(sheet, "B3", "10");
   setValue(sheet, "A4", "fixed:");
   setValue(sheet, "B4", "30");
   setValue(sheet, "A5", "disaggregate:");
   setValue(sheet, "B5", "1");

   setValue(sheet, "A8", "Result computed by the OPL model:");
   setValue(sheet, "A9", "State:");
   setValue(sheet, "A10", "Objective:");
   }
```

7. Add the code for the **Solve** button to launch the OPL solve process.

```
private void SolveButton_Click(Office.CommandBarButton Ctrl,
                                                        ref bool
CancelDefault)
{
   Excel.Worksheet sheet =
               (Excel.Worksheet)this.Application.ActiveWorkbook.
ActiveSheet;
   System.IO.StringWriter errHandlerErrors = new System.IO.StringWriter
();
      try
      {
          setValue(sheet, "B9", "Computing...");
          SolveProblem(sheet, errHandlerErrors);
          setValue(sheet, "B9", "Success");
      }
          catch (Exception ex)
```

```
      {
          setValue(sheet, "B9", "Error:" + ex.ToString()
                                + "(" + errHandlerErrors.ToString() + ")")
;
      }
}

private void SolveProblem(Excel.Worksheet sheet,
                                        System.IO.StringWriter
errHandlerErrors)
{
// actually solve the problem with OPL
}
```

8. Add some references to the OPL .NET APIs used to solve the problem.
   a. In the Solution view, select the project (ExcelWarehouse), the References, and click **Add Reference**.
   b. In the **Add Reference Wizard**, select **Browse**, navigate to the OPL installation, and add `<OPL>/lib/oplall.dll`.
   c. At the top of the `ThisAddIn.cs` file, append the following lines to the `using` list.

```
using ILOG.Concert;
using ILOG.CPLEX;
using ILOG.OPL;
using Exception = System.Exception;
```

9. Add the code that actually solves the problem using OPL.

```
private void SolveProblem(Excel.Worksheet sheet,
                                        System.IO.StringWriter
errHandlerErrors)
{
   OplFactory.DebugMode = true;
   OplFactory oplF = new OplFactory();
   OplErrorHandler errHandler =
                              oplF.CreateOplErrorHandler(errHandlerErrors)
;
   OplModelSource modelSource =
         oplF.CreateOplModelSourceFromString(GetModelText(), "warehouse")
;
   OplSettings settings = oplF.CreateOplSettings(errHandler);
   OplModelDefinition def = oplF.CreateOplModelDefinition
                                                    (modelSource, settings)
;
   Cplex cplex = oplF.CreateCplex();
   OplModel opl = oplF.CreateOplModel(def, cplex);
   OplDataSource dataSource = new MyParams(oplF, sheet);
   opl.AddDataSource(dataSource);
   opl.Generate();
   if (cplex.Solve())
```

```
   {
      setValue(sheet, "B10", "" + opl.Cplex.ObjValue);
   }
   else
   {
      setValue(sheet, "B10", "No solution");
   }
   oplF.End();
}

/**
* This class feeds data to the OPL model from the appropriate cells of
the
* input Excel worksheet.
*/
class MyParams : CustomOplDataSource
{
   private Excel.Worksheet _xlDataSheet;

   public MyParams(OplFactory oplF, Excel.Worksheet xlDataSheet): base
(oplF)
   {
      _xlDataSheet = xlDataSheet;
   }
```

```
   public override void CustomRead()
   {
        OplDataHandler handler = this.DataHandler;

        try
        {
           handler.StartElement("nbWarehouses");
           handler.AddIntItem(Int16.Parse(getValue(_xlDataSheet, "B2")))
;
           handler.EndElement();

           handler.StartElement("nbStores");
           handler.AddIntItem(Int16.Parse(getValue(_xlDataSheet, "B3")))
;
           handler.EndElement();

           handler.StartElement("fixed");
           handler.AddIntItem(Int16.Parse(getValue(_xlDataSheet, "B4")))
;
           handler.EndElement();

           handler.StartElement("disaggregate");
           handler.AddIntItem(Int16.Parse(getValue(_xlDataSheet, "B5")))
;
           handler.EndElement();
         }
```

```
         catch (Exception ex)
         {
             MessageBox.Show("Exception in custom data source:
                                                              " + ex.ToString
());
          }
   }
}
```

```
/**
* This is the warehouse OPL model:
*/
static String GetModelText()
{
   String model = "";
   model += "int   fixed        = ...;";
   model += "int   nbWarehouses = ...;";
   model += "int   nbStores     = ...;";
   model += "int   disaggregate = ...;";
   model += "assert nbStores > nbWarehouses;";

   model += "range Warehouses = 1..nbWarehouses;";
   model += "range Stores     = 1..nbStores;";

   model += "int capacity[w in Warehouses] = nbStores div nbWarehouses
                                      + w mod (nbStores div nbWarehouses)
;";
   model += "int supplyCost[s in Stores][w in Warehouses] =
                                                    1+((s+10*w) mod 100)
;";

   model += "dvar float open[Warehouses] in 0..1;";
   model += "dvar float supply[Stores][Warehouses] in 0..1;";

   model += "minimize ";
   model += "sum(w in Warehouses) fixed * open[w] +";
   model += "sum(w in Warehouses, s in Stores) supplyCost[s][w]
                                                          * supply[s][w]
;";

   model += "constraints {";
   model += "  forall(s in Stores)";
   model += "    sum(w in Warehouses) supply[s][w] == 1;";
   model += "  forall(w in Warehouses)";
   model += "    sum(s in Stores) supply[s][w] <= open[w]*capacity[w];";

   model += "  if (disaggregate == 1) {";
   model += "   forall(w in Warehouses, s in Stores)";
   model += "      supply[s][w] <= open[w];";
   model += "  }";
   model += "}";
```

```
    return model;
}
```

The code is complete.

10. Launch Microsoft Excel with your plug-in, using the Visual Studio **Debug/Start Debugging** command.

    Excel starts, with the new command bar containing OPL Warehouse Setup and OPL Warehouse Solve.

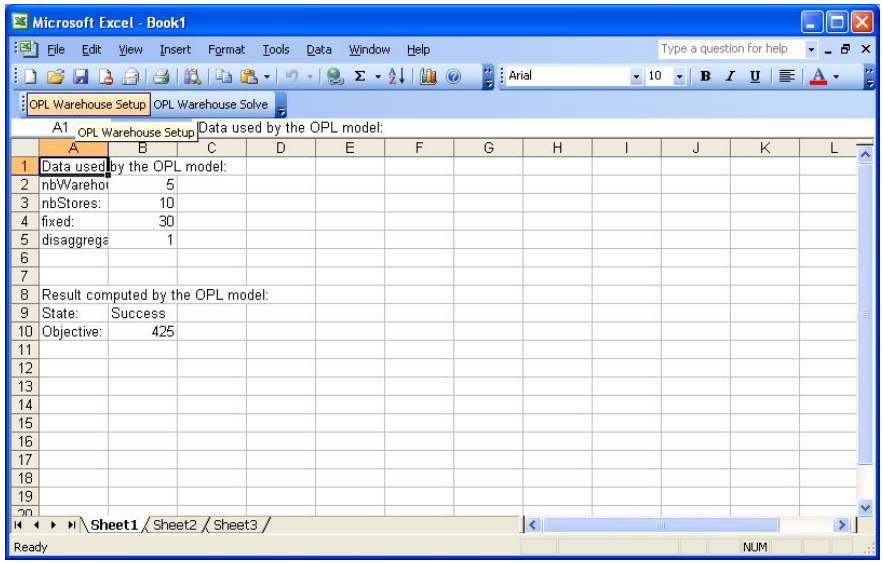
    a. Click the button **OPL Warehouse Setup**.

       It populates the current sheet with default values for the problem.

    b. Edit the values as appropriate.

    c. Click the button **OPL Warehouse Solve**.

       It solves the problem using OPL, displaying results in cell B10 (425 for the default values).



Integrating OPL .NET API into an MS Excel worksheet

Useful training material on working with VSTO 2005 is available at:

http://msdn2.microsoft.com/en-us/library/ebax1172(VS.80).aspx

Memory management

The recommended way to manage memory in Concert applications is to use the method `IloEnv.end` to clear all the memory currently in use.

If you need more control on the memory used by your OPL objects, `IloOplModel` objects offer the method `end`.

In the default case, after an OPL model instance has been ended, all its Concert objects that correspond to data elements are still available. After the method `end`, no objects remain. *Availability of Concert objects* summarizes the availability of Concert objects accessed through OPL elements.

Availability of Concert objects

OPL Element	Definition	Available after end()	Available a
external data	elements read from a data source	no	yes
internal data	elements initialized inside the .mod file	no	yes
dvar array solution value	An array with values of decision variables for the current solution, available by calling the method asIntMap() or asNumMap() for a dvar array element	no	no
postprocessing	All elements declared for postprocessing	no	no

When postprocess is called multiple times, as when processing intermediate feasible solutions, the second call ends the objects created for the first call.

See these two stock-cutting examples:

<OPL_dir>\examples\opl\cutstock\cutstock_main.mod

<OPL_dir>\examples\opl\cutstock\cutstock_int_main.mod

where `<OPL_dir>` is your installation directory.

Index