Java tutorial

Applications written in the Java programming language use CPLEX with Concert Technology in the Java API.

- Overview
  A typical application with CPLEX in Java includes these features.
- Compiling CPLEX in Concert Technology Java applications
  When you compile CPLEX in a Java application, you need to specify the location of the CPLEX JAR in the classpath.
- The design of CPLEX in Concert Technology Java applications
  Your Java application includes CPLEX as a component.
- The anatomy of a Concert Technology Java application
  A Java application of CPLEX includes these parts.
- Building and solving a small LP model in Java
  An example shows how to solve a model in a Java application of CPLEX.

Parent topic: Tutorials

Overview

A typical application with CPLEX in Java includes these features.
Concert Technology allows your application to call IBM ILOG CPLEX directly, through the Java Native Interface (JNI). This Java interface supplies a rich means for you to use Java objects to build your optimization model.
The class IloCplex implements the Concert Technology interface for creating variables and constraints. It also provides functionality for solving Mathematical Programing (MP) problems and accessing solution information.

Parent topic: Java tutorial

Compiling CPLEX in Concert Technology Java applications

When you compile CPLEX in a Java application, you need to specify the location of the CPLEX JAR in the classpath.

- Paths and JARs
  Your Java classpath is important to CPLEX in Java applications.
- Adapting build procedures to your platform
  CPLEX makefiles and other aids support Java application development with CPLEX.
- In case problems arise
  CPLEX supports trouble-shooting procedures specific to Java applications.

Parent topic: Java tutorial

Paths and JARs

Your Java classpath is important to CPLEX in Java applications.

When compiling a Java application that uses Concert Technology, you need to inform the Java compiler where to find the file cplex.jar containing the CPLEX Concert Technology class library. To do this, you add the cplex.jar file to your classpath. This is most easily done by passing the command-line option to the Java compiler javac, like this:

–classpath *path_to_*cplex.jar

If you need to include other Java class libraries, you should add the corresponding jar files to the classpath as well. Ordinarily, you should also include the current directory (.) to be part of the Java classpath.

At execution time, the same classpath setting is needed. Additionally, since CPLEX is implemented via JNI, you need to instruct the Java Virtual Machine (JVM) where to find the shared library (or dynamic link library) containing the native code to be called from Java. You indicate this location with the command line option:

-Djava.library.path=*path_to_shared_library*

to the java command. Note that, unlike the cplex.jar file, the shared library is system-dependent; thus the exact path name of the location for the library to be used may differ depending on the platform you are using.

Parent topic: Compiling CPLEX in Concert Technology Java applications

Adapting build procedures to your platform

CPLEX makefiles and other aids support Java application development with CPLEX.

About this task

Pre-configured compilation and runtime commands are provided in the standard distribution, through the UNIX makefiles and Windows javamake file for Nmake . However, these scripts presume a certain relative location for the files already mentioned; for application development, most users will have their source files in some other location.
Here are suggestions for establishing build procedures for your application.

Procedure

1. First check the readme.html file in the standard distribution, under the Supported Platforms heading to locate the *machine* and *libformat* entry for your UNIX platform, or the compiler and library-format combination for Windows.
2. Go to the subdirectory in the examples directory where CPLEX is installed on your machine. On UNIX, this will be *machine/libformat*, and on Windows it will be *compiler\libformat*. This subdirectory will contain a makefile or javamake appropriate for your platform.
3. Then use this file to compile the examples that came in the standard distribution by calling make execute_java (UNIX) or nmake -f javamake execute (Windows).
4. Carefully note the locations of the needed files, both during compilation and at run time, and convert the relative path names to absolute path names for use in your own working environment.

Parent topic: Compiling CPLEX in Concert Technology Java applications

In case problems arise

CPLEX supports trouble-shooting procedures specific to Java applications.

About this task

If a problem occurs in the compilation phase, make sure your Java compiler is correctly set up and that your classpath includes the cplex.jar file.
If compilation is successful and the problem occurs when executing your application, there are three likely causes:

Procedure

1. If you get a message like java.lang.NoClassDefFoundError your classpath is not correctly set up. Make sure you use -classpath <path_to_cplex.jar> in your java command.
2. If you get a message like java.lang.UnsatisfiedLinkError, you need to set up the path correctly so that the JVM can locate the CPLEX shared library. Make sure you use the following option in your java command:-Djava.library.path=<path_to_shared_library>

Parent topic: Compiling CPLEX in Concert Technology Java applications

The design of CPLEX in Concert Technology Java applications

Your Java application includes CPLEX as a component.

Figure 1. A View of CPLEX in Concert Technology

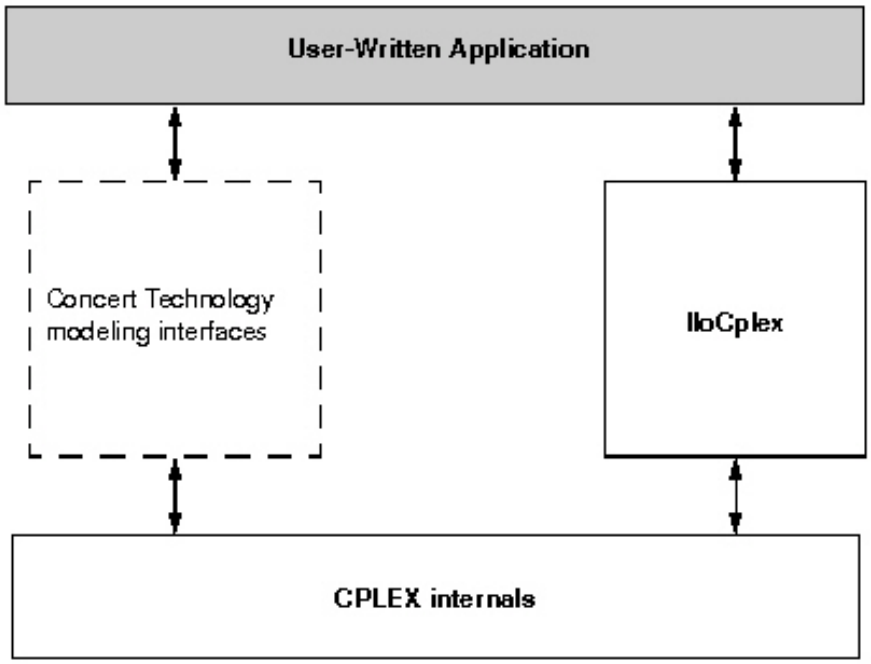

Figure 1 illustrates the design of Concert Technology and how a user-application uses it. Concert Technology defines a set of interfaces for modeling objects. Such interfaces do not actually consume memory. (For this reason, the box in the figure has a dotted outline.) When a user creates a Concert Technology modeling object using CPLEX, an object is created in CPLEX to implement the interface defined by Concert Technology. However, a user application never accesses such objects directly but only communicates with them through the interfaces defined by Concert Technology.

For more detail about these ideas, see the *CPLEX User's Manual*, especially the topic Concert Technology for Java users.

Parent topic: Java tutorial

The anatomy of a Concert Technology Java application

A Java application of CPLEX includes these parts.

- Structure of an application
  Java applications using CPLEX observe object-oriented conventions.
- Create the model
  Java methods create a model in a Java application of CPLEX.
- Solve the model
  Java methods create the object-oriented optimizer in a Java application of CPLEX.
- Query the results
  Java methods query results from a Java application of CPLEX.

Parent topic: Java tutorial

Structure of an application

Java applications using CPLEX observe object-oriented conventions.
To use the CPLEX Java interfaces, you need to import the appropriate packages into your application with these lines:

```
import ilog.concert.*;
import ilog.cplex.*;
```

As for every Java application, a CPLEX application is implemented as a method of a class. In this discussion, the method will be the static main method. The first task is to create an IloCplex object. It is used to create all the modeling objects needed to represent the model. For example, an integer variable with bounds 0 and 10 is created by calling cplex.intVar(0, 10) , where cplex is the IloCplex object.
Since Java error handling in CPLEX uses exceptions, you should include the Concert Technology part of an application in a try /catch statement. All the exceptions thrown by any Concert Technology method are derived from IloException. Thus IloException should be caught in the catch statement.
In summary, here is the structure of a Java application that calls CPLEX:

```
-classpath <path_to_cplex.jar>


-Djava.library.path=<path_to_shared_library>
import ilog.concert.*;
import ilog.cplex.*;
 import ilog.concert.*;
 import ilog.cplex.*;
 static public class Application {
   static public main(String[] args) {
      try {
        IloCplex cplex = new IloCplex();
        // create model and solve it
      } catch (IloException e) {
         System.err.println("Concert exception caught: " + e);
      }
    }
  }
```

Parent topic: The anatomy of a Concert Technology Java application

Create the model

Java methods create a model in a Java application of CPLEX.
The IloCplex object provides the functionality to create an optimization model that can be solved with IloCplex. The class IloCplex implements the Concert Technology interface IloModeler and its extensions IloMPModeler and IloCplexModeler. These interfaces define the constructors for modeling objects of the following types, which can be used with IloCplex :

Modeling classes	Description
IloNumVar	modeling variables
IloRange	ranged constraints of the type lb <= expr <= ub
IloObjective	optimization objective
IloNumExpr	expression using variables

Modeling variables are represented by objects implementing the IloNumVar interface defined by Concert Technology. Here is how to create three continuous variables, all with bounds 0 and 100 :

```
IloNumVar[] x = cplex.numVarArray(3, 0.0, 100.0);
```

There is a wealth of other methods for creating arrays or individual modeling variables. The documentation for IloModeler, IloCplexModeler, and IloMPModeler gives you the complete list.
Modeling variables build expressions, of type IloNumExpr, for use in constraints or the objective function of an optimization model. For example, the expression:

```
x[0] + 2*x[1] + 3*x[2]
```

can be created like this:

```
IloNumExpr expr = cplex.sum(x[0],
                            cplex.prod(2.0, x[1]),
                            cplex.prod(3.0, x[2]));
```

Another way of creating an object representing the same expression is to use an expression of IloLinearNumExpr. Here is how:

```
  IloLinearNumExpr expr = cplex.linearNumExpr();
 expr.addTerm(1.0, x[0]);
 expr.addTerm(2.0, x[1]);
 expr.addTerm(3.0, x[2]);
```

The advantage of using IloLinearNumExpr over the first way is that you can more easily build up your linear expression in a loop, which is what is typically needed in more complex applications. The interface IloLinearNumExpr is an extension of IloNumExpr and thus can be used anywhere an expression can be used.
As mentioned before, expressions can be used to create constraints or an objective function for a model. Here is how to create a minimization objective for that expression:
In addition to your creating an objective, you must also instruct IloCplex to use that objective in the model it solves. To do so, *add* the objective to IloCplex like this:

```
cplex.add(obj);
```

Every modeling object that is to be used in a model must be added to the IloCplex

object. The variables need not be explicitly added as they are treated implicitly when used in the expression of the objective. More generally, every modeling object that is referenced by another modeling object which itself has been added to IloCplex, is implicitly added to IloCplex as well.
There is a shortcut notation for creating and adding the objective:
cplex.addMinimize(expr);This shortcut uses the method addMinimize
Since the objective is not otherwise accessed, it does not need to be stored in the variable obj .
Adding constraints to the model is just as easy. For example, the constraint
-x[0] + x[1] + x[2] <= 20.0
can be added by calling:

```
cplex.addLe(cplex.sum(cplex.negative(x[0]), x[1], x[2]), 20);
```

Again, many methods are provided for adding other constraint types, including equality constraints, greater than or equal to constraints, and ranged constraints. Internally, they are all represented as IloRange objects with appropriate choices of bounds, which is why all these methods return IloRange objects. Also, note that the previous expressions could have been created in many different ways, including the use of IloLinearNumExpr.

Parent topic: The anatomy of a Concert Technology Java application

Solve the model

Java methods create the object-oriented optimizer in a Java application of CPLEX. So far you have seen some methods of IloCplex for creating models. All such methods are defined in the interfaces IloModeler and its extension IloMPModeler and IloCplexModeler. However, IloCplex not only implements these interfaces but also provides additional methods for solving a model and querying its results.

After you have created a model as explained in Create the model, the object IloCplex is ready to solve the problem, which consists of the model and all the modeling objects that have been added to it. Invoking the optimizer then is as simple as calling the method solve .

That method returns a Boolean value indicating whether the optimization succeeded in finding a solution. If no solution was found, false is returned. If true is returned, then CPLEX found a feasible solution, though it is not necessarily an optimal solution. More precise information about the outcome of the last call to the method solve can be obtained from the method getStatus .

The returned value tells you what CPLEX found out about the model: whether it found the optimal solution or only a feasible solution, whether it proved the model to be unbounded or infeasible, or whether nothing at all has been proved at this point. Even more detailed information about the termination of the optimizer call is available through the method getCplexStatus .

Parent topic: The anatomy of a Concert Technology Java application

Query the results

Java methods query results from a Java application of CPLEX.

If the method solve succeeded in finding a solution, you will then want to access that solution. The objective value of that solution can be queried using a statement like this:

```
double objval = cplex.getObjValue();
```

Similarly, solution values for all the variables in the array x can be queried by calling:

```
double[] xval = cplex.getValues(x);
```

More solution information can be queried from IloCplex , including slacks and, depending on the algorithm that was applied for solving the model, duals, reduced cost information, and basis information.

Parent topic: The anatomy of a Concert Technology Java application

Building and solving a small LP model in Java

An example shows how to solve a model in a Java application of CPLEX.

- Example: LPex1.java
  This example illustrates solving a model in a Java application of CPLEX.
- Modeling by rows
  Java methods support modeling by rows in this example of a Java application of CPLEX.
- Modeling by columns
  Java methods support modeling by columns in this example of a Java application of CPLEX.
- Modeling by nonzeros
  Java methods support modeling by nonzeros in this example of a Java application of CPLEX.

Parent topic: Java tutorial

Example: LPex1.java

This example illustrates solving a model in a Java application of CPLEX.
The example LPex1.java , part of the standard distribution of CPLEX, is a program that builds a specific small LP model and then solves it. This example follows the general structure found in many CPLEX Concert Technology applications, and demonstrates three main ways to construct a model:
- Modeling by rows;
- Modeling by columns;
- Modeling by nonzeros.

Example LPex1.java is an extension of the example presented in Entering the example:

Maximize	$x1 + 2x2 + 3x3$
subject to	$-x1 + x2 . + x3 \ 20$ $x1 - 3x2 + x3 \ 30$
with these bounds	$0 \le x1 \le 40$ $0 \le x2 \le infinity$ $0 \le x3 \le infinity$

After an initial check that a valid option string was provided as a calling argument, the program begins by enclosing all executable statements that follow in a try/catch pair of statements. In case of an error CPLEX Concert Technology will throw an exception of type IloException, which the catch statement then processes. In this simple example, an exception triggers the printing of a line stating Concert exception ‘e’ caught , where e is the specific exception.
First, create the model object cplex by executing the following statement:

```
IloCplex cplex = new IloCplex();
```

At this point, the cplex object represents an empty model, that is, a model with no variables, constraints, or other content. The model is then populated in one of several ways depending on the command line argument. The possible choices are implemented in the methods
- populateByRow
- populateByColumn
- populateByNonzero

All these methods pass the same three arguments. The first argument is the cplex object to be populated. The second and third arguments correspond to the variables (var) and range constraints (rng) respectively; the methods will write to var[0] and rng[0] an array of all the variables and constraints in the model, for later access.
After the model has been created in the cplex object, it is ready to be solved by a call to cplex.solve. The solution log will be output to the screen; this is because IloCplex prints all logging information to the OutputStreamcplex. output, which by default is initialized to System.out. You can change this by calling the method cplex. setOut. In particular, you can turn off logging by setting the output stream to null, that is, by calling cplex.setOut(null). Similarly, IloCplex issues warning messages to cplex. warning, and cplex. setWarning can be used to change (or turn off) the OutputStream that will be used.
If the solve method finds a feasible solution for the active model, it returns true. The next section of code accesses the solution. The method cplex. getValues(var[0])

returns an array of primal solution values for all the variables. This array is stored as double[]x . The values in x are ordered such that x[j] is the primal solution value for variable var[0][j] . Similarly, the reduced costs, duals, and slack values are queried and stored in arrays dj, pi, and slack, respectively. Finally, the solution status of the active model and the objective value of the solution are queried with the methods IloCplex. getStatus and IloCplex. getObjValue, respectively. The program then concludes by printing the values that have been obtained in the previous steps, and terminates after calling cplex. end to free the memory used by the model object; the catch method of IloException provides screen output in case of any error conditions along the way.
The remainder of the example source code is devoted to the details of populating the model object and the following three sections provide details on how the methods work.
You can view the complete program online in the standard distribution of the product at *yourCPLEXinstallation*/examples/src/LPex1.java.

Parent topic: Building and solving a small LP model in Java

Modeling by rows

Java methods support modeling by rows in this example of a Java application of CPLEX.

The method populateByRow creates the model by adding the finished constraints and objective function to the active model, one by one. It does so by first creating the variables with the method cplex. numVarArray. Then the minimization objective function is created and added to the active model with the method IloCplex. addMinimize. The expression that defines the objective function is created by a method, IloCplex. scalProd, that forms a scalar product using an array of objective coefficients times the array of variables. Finally, each of the two constraints of the model are created and added to the active model with the method IloCplex. addLe. For building the constraint expression, the methods IloCplex. sum and IloCplex. prod are used, as a contrast to the approach used in constructing the objective function.

Parent topic: Building and solving a small LP model in Java

Modeling by columns

Java methods support modeling by columns in this example of a Java application of CPLEX.

While for many examples population by rows may seem most straightforward and natural, there are some models where population by columns is a more natural or more efficient approach to implement. For example, problems with network structure typically lend themselves well to modeling by column. Readers familiar with matrix algebra may view the method populateByColumn as producing the transpose of what is produced by the method populateByRow . In contrast to modeling by rows, modeling by columns means that the coefficients of the constraint matrix are given in a column-wise way. As each column represents the constraint coefficients for a given variable in the linear program, this modeling approach is most natural where it is easy to access the matrix coefficients by iterating through all the variables, such as in network flow problems.

Range objects are created for modeling by column with only their lower and upper bound. No expressions are given; building them at this point would be impossible since the variables have not been created yet. Similarly, the objective function is created only with its intended optimization sense, and without any expression.

Next the variables are created and installed in the existing ranges and objective. These newly created variables are introduced into the ranges and the objective by means of column objects, which are implemented in the class IloColumn. Objects of this class are created with the methods IloCplex. column, and can be linked together with the method IloColumn. and to form aggregate IloColumn objects.

An instance of IloColumn created with the method IloCplex. column contains information about how to use this column to introduce a new variable into an existing modeling object. For example, if obj is an instance of a class that implements the interface IloObjective, then cplex.column(obj, 2.0) creates an instance of IloColumn containing the information to install a new variable in the expression of the IloObjective object obj with a linear coefficient of 2.0. Similarly, for rng, a constraint that is an instance of a class that implements the interface IloRange, the invocation of the method cplex.column(rng, -1.0) creates an IloColumn object containing the information to install a new variable into the expression of rng , as a linear term with coefficient -1.0.

When you use the approach of modeling by column, new columns are created and installed as variables in all existing modeling objects where they are needed. To do this with Concert Technology, you create an IloColumn object for every modeling object in which you want to install a new variable, and link them together with the method IloColumn. and . For example, the first variable in populateByColumn is created like this:

The three methods model.column create IloColumn objects for installing a new variable in the objective obj and in the constraints r0 and r1, with linear coefficients 1.0 , -1.0, and 1.0, respectively. They are all linked to an aggregate column object by the method and. This aggregate column object is passed as the first argument to the method numVar, along with the bounds 0.0 and 40.0 as the other two arguments. The method numVar now creates a new variable and immediately installs it in the modeling objects obj, r0, and r1 as defined by the aggregate column object. After it has been installed, the new variable is returned and stored in var[0][0] .

Parent topic: Building and solving a small LP model in Java

Modeling by nonzeros

Java methods support modeling by nonzeros in this example of a Java application of CPLEX.

The last of the three functions for building the model is populateByNonzero . This function creates the variables with only their bounds, and the empty constraints, that is, ranged constraints with only lower and upper bound but with no expression. Only after that are the expressions constructed over these existing variables, in a manner similar to the ones already described; they are installed in the existing constraints with the method IloRange. setExpr .

Parent topic: Building and solving a small LP model in Java