In the name of God

Part 3. ILOG CPLEX

3.4. CPLEX Java Applications

Spring 2010

Instructor: Dr. Masoud Yaghini

Outline

- Architecture of a CPLEX Java Application
- Compiling CPLEX Java Applications
- Solving the Model
- Accessing Solution Information
- Modeling by Column
- References

Architecture of a CPLEX Java Application

Architecture of a CPLEX Java Application

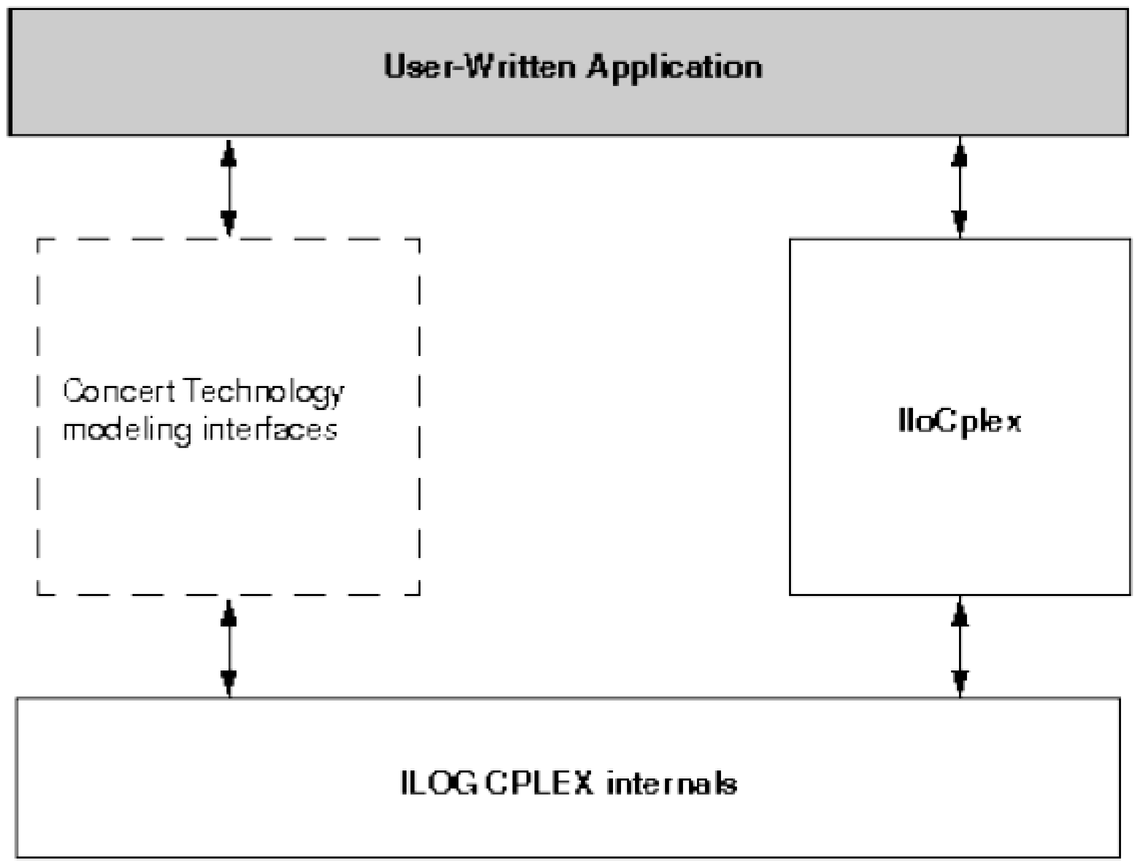
- **ILOG Concert Technology**
  - allows your application to call ILOG CPLEX directly, through the **Java Native Interface (JNI)**.
  - This Java interface supplies a rich means for you to use Java objects to build your optimization model.
- **IloCplex** class implements the **ILOG Concert Technology interface** for:
  - Creating variables and constraints
  - Providing functionality for solving Mathematical Programing (MP) problems
  - Accessing solution information

Architecture of a CPLEX Java Application

- **ILOG Concert Technology interface**
  - For example, every variable in a model is represented by an object that implements the Concert Technology variable interface IloNumVar .
  - The user code accesses the variable only through its Concert Technology interface.
  - Similarly, all other modeling objects are accessed only through their respective Concert Technology interfaces from the user-written application, while the actual objects are maintained in the ILOG CPLEX database.

Architecture of a CPLEX Java Application

- **A view of Concert Technology for Java users**



Architecture of a CPLEX Java Application

- **The ILOG CPLEX internals**
  - Include the computing environment, its communication channels, and your problem objects.

Architecture of a CPLEX Java Application

- **Creating a Java application:**
  - Create a model of your problem
  - Solve the model
  - Accessing solution information
  - Modifying the model explains

Architecture of a CPLEX Java Application

- To use the ILOG CPLEX Java interfaces, you need to import the appropriate packages into your application, using:

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

Architecture of a CPLEX Java Application

- The structure of a Java application that calls 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);
       }
     }
   }
```

Compiling CPLEX Java Applications

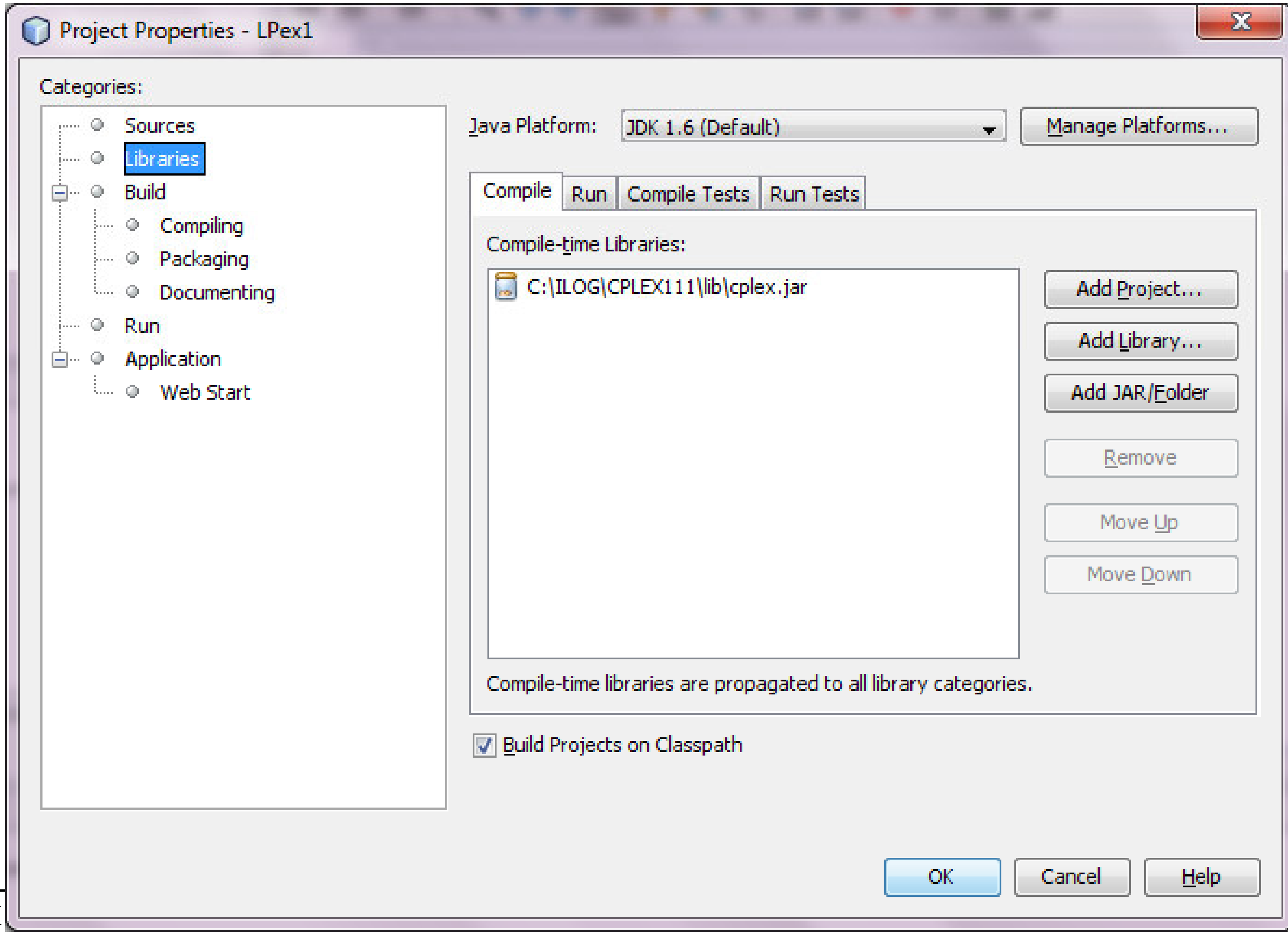
Compiling CPLEX Java Applications

- **cplex.jar**
  - containing the CPLEX Concert Technology class library.
- When compiling a Java application that uses ILOG Concert Technology, you need to inform the **Java compiler** where to find the file **cplex.jar**
- You need to set up the path correctly so that the **JVM** can locate the CPLEX shared library.

-Djava.library.path=..\..\..\bin\x86_win32

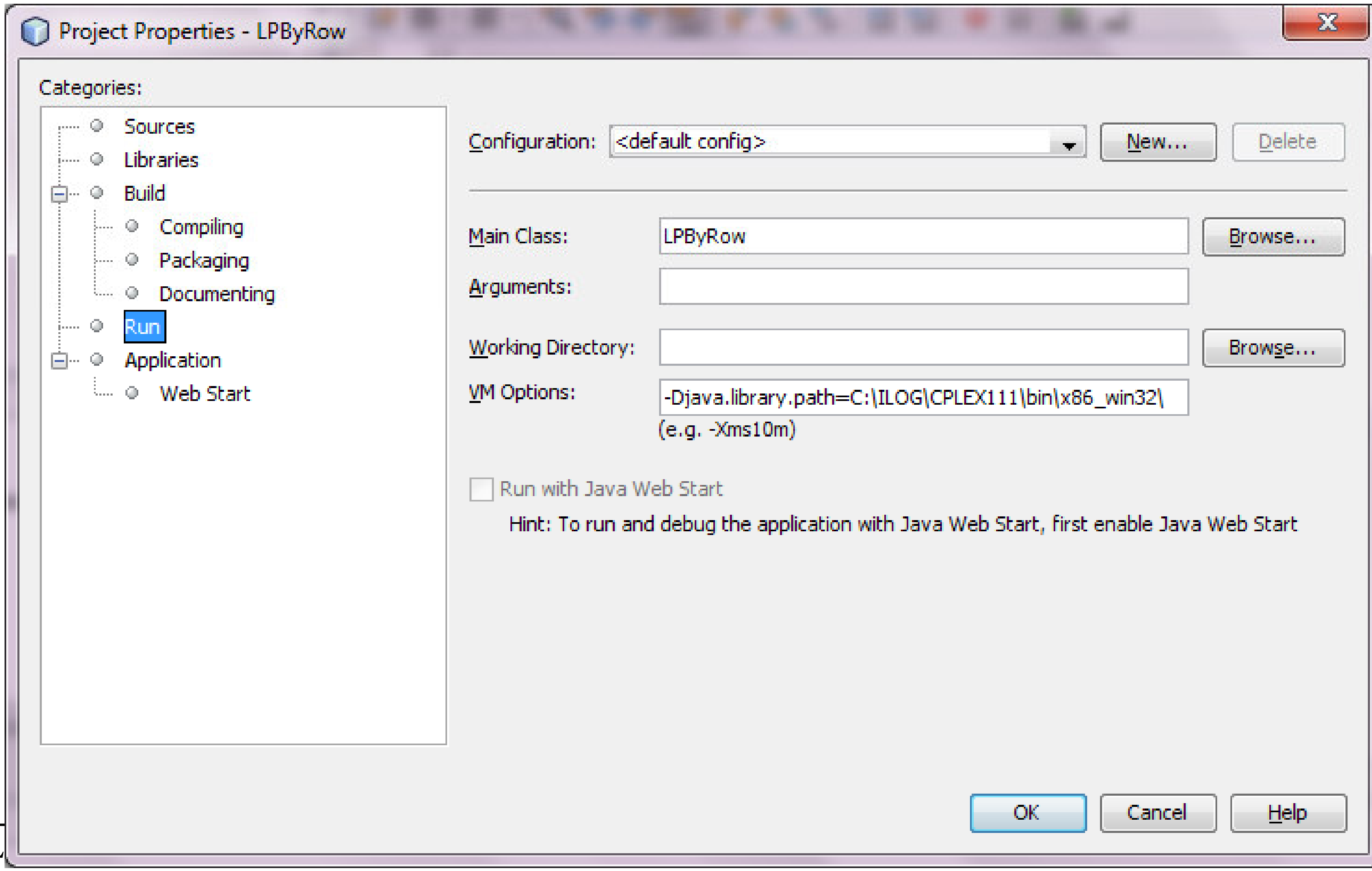
Compiling CPLEX Java Applications

- Add cplex.jar in NetBeans



Compiling CPLEX Java Applications

- Set up the path correctly so that the **JVM** in NetBeans



Modeling an Optimization Problem

Modeling an Optimization Problem

- **Classes:**
  - IloCplexModeler class
  - IloCplex class
- **Interfaces:**
  - IloModeler
  - IloMPModeler
    - IloMPModeler extends IloModeler
  - IloCPModeler
    - IloCPModeler extends IloModeler

Modeling an Optimization Problem

- IloCplex class
  - The class IloCplex extends IloCplexModeler.
  - All the **modeling methods** in IloCplex derive from IloCplexModeler.
  - IloCplex implements the **solving methods**.
  - IloCplex implements these interfaces IloModeler and IloMPModeler

Modeling an Optimization Problem

- Modeling objects are created using methods of an instance of IloModeler or one of its extensions, such as IloMPModeler or IloCPModeler .

Modeling an Optimization Problem

- Model will be an instance of IloCplex , and it is created like this:

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

- Since class IloCplex implements IloMPModeler (and thus its parent interface IloModeler) all methods from IloMPModeler and IloModeler can be used for building a model.

Modeling an Optimization Problem

- IloModeler defines the methods to:
  - create modeling variables of type integer, floating-point, or Boolean
  - construct simple expressions using modeling variables
  - create objective functions
  - create ranged constraints, that is, constraints of the form:

Variables in a model

- A modeling variable is represented by an object of type IloNumVar or one of its extensions.
- An example of the method is:

  ```
  IloNumVar  x = cplex.numVar(lb, ub, IloNumVarType.Float, "xname");
  ```

  - This constructor method allows you to set all the attributes of a variable: its **lower** and **upper bounds**, its **type**, and its **name**.
- intVar() method
  - To create integer variables
- boolVar() method
  - To create 0 / 1 variables

Modeling an Optimization Problem

- numVarArray(), intVarArray(), boolVarArray() Methods
  - methods for creating a complete array of modeling variables at one time.

Expressions

- **Expressions**
  - Modeling variables are typically used in **expressions** that define constraints or objective functions.
  - Expressions are represented by objects of type IloNumExpr
  - They are built using methods such as sum, prod, diff , negative, and square.
  - For example, the expression:

    x1 + 2*x2

  - where x1 and x2 are IloNumVar objects, is constructed by this call:

```
IloNumExpr expr = cplex.sum(x1, cplex.prod(2.0, x2));
```

Ranged constraints

- **Ranged constraints**
  - are constraints of the form: lb ≤ expression ≤ ub
  - They are represented by objects of type IloRange
- The most general constructor is:

```
IloRange rng = cplex.range(lb, expr, ub, name);
```

  - lb and ub are double values,
  - expr is of type IloNumExpr
  - name is a string.

Ranged constraints

- **Ranged constraints** can be used to model any of the more commonly found constraints of the form:

  expr relation rhs

  - where relation is the relation =, ≤, or ≥.
- The following table shows how to choose lb and ub for modeling these relations:

relation	lb	ub	method
=	rhs	rhs	eq
≤	-Double.MAX_VALUE	rhs	le
≥	rhs	Double.MAX_VALUE	ge

Ranged constraints

- The last column contains the method to use directly to create the appropriate ranged constraint.
- For example, the constraint $expr \le 1.0$ is created by the call:

  `IloRange le = cplex.le(expr, 1.0);`
- Again, all constructors for ranged constraints come in pairs, one constructor with and one without an argument for the name.

The objective function

- The objective function is represented by an object of type IloObjective.
- Such objects are defined by:
  - **an optimization sense**: is represented by an object of class IloObjectiveSense, and can take two values,
    - IloObjectiveSense.Maximize
    - IloObjectiveSense.Minimize
  - **an expression**: is represented by an IloNumExpr
  - **an optional name**: is a string

The objective function

- For convenience, the methods maximize and minimize are provided to create a maximization or minimization objective respectively, without using an IloObjectiveSense parameter, for example:

  cplex.maximize(expr);

The active model

- The **active model** is the model implemented by the IloCplex object itself.
- The **constraints** and **objective functions** must be created and added to the active model.
- To facilitate this, for most constructors with a name such as ConstructorName, there is also a method addConstructorName which immediately adds the newly constructed modeling object to the active model.
- For example:

```
IloObjective obj = cplex.addMaximize(expr);
```

- is equivalent to:

```
IloObjective obj = cplex.add(cplex.maximize(expr));
```

Diet problem

- **Diet problem**
  - consists of finding the least expensive diet using a set of foods such that all nutritional requirements are satisfied.
- The example
  - foodCost[j]: a unit cost of food j
  - foodMin[j] & foodMax[j]: minimum and maximum amount of food j which can be used in the diet
  - nutrPerFood[i][j]: a nutritional value food j for nutrients i
  - nutrMin[i] & nutrMax[i] : in the diet the amount of every nutrient i consumed must be within these bounds
  - Buy[j] : the amount of food j to buy for the diet.

Diet problem

- Then the objective is:

  minimize Σ_j (Buy[j] * foodCost[j])

- The nutritional requirements, for all i :

  nutriMin[i] $\leq \Sigma_j$ nutrPerFood[i][j] * Buy[j] $\leq$ nutriMax[i]

- Every food must be within its bounds, for all j :

  foodMin[j] $\leq$ Buy[j] $\leq$ foodMax[j]

- The diet program:
  - Diet.java

Diet problem

- The example accepts a filename and two options -c and -i as command line arguments.
- Option -i allows you to create a MIP model where the quantities of foods to purchase must be integers.
- Option -c can be used to build the model by columns.

Diet problem

- The program starts by evaluating the command line arguments and reading the input data file.
- The input data of the diet problem is read from a file using an object of the embedded class Diet.Data .
- Its constructor requires a file name as an argument.
- Using the class InputDataReader , it reads the data from that file.
- This class does not use ILOG CPLEX or Concert Technology in any special way.

Diet problem

- **Exception handling**
  - In case of an error, ILOG CPLEX will throw an exception of type IloException or one of its subclasses.
  - Thus the entire ILOG CPLEX program is enclosed in try/catch statements.
  - The InputDataReader can throw exceptions of type java.io.IOException or InputDataReader.InputDataReaderException

Diet problem

- cplex.end
  - The call to the method cplex.end frees the memory that ILOG CPLEX uses.

buildModelByRow Method

- The method accepts several arguments.
- model
  - is used for two purposes:
    - creating other modeling objects
    - representing the model being created
- data
  - contains the data for the model to be built.
- Buy
  - containing the model's variables
- type
  - type of the variables being created

buildModelByRow Method

- **Creating the modeling variables**
  - The method creates variables one by one, and storing them in array Buy .
  - Each variable j is initialized to have bounds data.foodMin[j] and data.foodMax[j] and to be of type type.
- **Constructing the objective function**
  - The variables are used to construct the objective function expression with the method: model.scalProd(foodCost, Buy)
  - This expression is immediately used to create the minimization objective which is directly added to the active model by addMinimize.

buildModelByRow Method

- Adding the nutritional constraints
  - For each nutrient i the expression representing the amount of nutrient in a diet with food levels Buy is computed using: model.scalProd(nutrPerFood[i], Buy)
  - This amount of nutrient must be within the ranged constraint bounds nutrMin[i] and nutrMax[i].
  - This constraint is created and added to the active model with addRange.

Solving the Model

Solving the Model

- After creating an optimization problem in your active model, you solve it by means of the IloCplex object.
- For an object named cplex , for example, you solve by calling the method like this:

  cplex.solve();
- The solve method returns a Boolean value specifying whether or not a feasible solution was found and can be queried.
- When true is returned, the solution that was found may not be the optimal one; for example, the optimization may have terminated prematurely because it reached an iteration limit.

Solving the Model

- Additional information about a possible solution can be queried with the method getStatus
- **Possible statuses:**
  - Error: an error occurred during the optimization.
  - Unknown: the active model far enough to prove anything about it. A common reason may be that a time limit was reached.
  - Feasible: A feasible solution for the model has been proven to exist.
  - Bounded: It has been proven that the active model has a finite bound in the direction of optimization. However, this does not imply the existence of a feasible solution.

Solving the Model

- **Possible statuses (cont.):**
  - Optimal: The active model has been solved to optimality. The optimal solution can be queried.
  - Infeasible: The active model has been proven to possess no feasible solution.
  - Unbounded: The active model has been proven to be unbounded. This does not include the notion that the model has been proven to be feasible.
  - Infeasible Or Unbounded: The active model has been proven to be infeasible or unbounded.

Accessing Solution Information

Accessing Solution Information

- If a solution has been found with the solve method, you access it.
- The objective function:

  double objval = cplex.getObjValue();

- The values of individual modeling variables:

  double x1 = cplex.getValue(var1);

- Solution values for an array of variables:

  double[] x = cplex.getValues(vars);

- You can query slack values for the constraints by: IloCplex.getSlack or IloCplex.getSlacks

Diet problem

- The diet program:
  - Diet.java

Exporting and Importing Models

- **Exporting models**
  - The method IloCplex.exportModel writes the active model to a file.
  - The format of the file depends on the file extension in the name of the file. For example:

    cplex.exportModel(“diet.lp");
- **Importing models**
  - A model can be read by means of the method IloCplex.importModel .
  - Both these methods are documented more fully in the reference manual of the Java API.

Dual Solution Information

- When solving an LP, all the algorithms also compute dual solution information .
- You can access reduced costs by calling the method IloCplex.getReducedCost or IloCplex.getReducedCosts
- You can access dual solution values for the ranged constraints by using the methods:

  IloCplex.getDual or

  IloCplex.getDuals .

Modeling by Column

Modeling by Column

- The concept of **modeling by column** modeling comes from the matrix view of mathematical programming problems.
- The columns of the constraint matrix correspond to variables.
- Modeling by column can be more generally understood as using columns to hold a place for new variables to install in modeling objects

Modeling by Column

- Individual IloColumn objects define how to install a new variable in one existing modeling object and are created with one of the IloMPModeler.column methods.
- Several IloColumn objects can be linked together (with the IloCplex.and method) to install a new variable in all modeling objects in which it is to appear.

Modeling by Column

- For example:

```
IloColumn col =
    cplex.column(obj,1.0).and(cplex.column(rng, 2.0));
```

  - This creates a new variable and install it in the objective function represented by obj with a linear coefficient of 1.0 and in the ranged constraint rng with a linear coefficient of 2.0 .
- After creating the proper column object, use it to create a new variable by passing it as the first parameter to the **variable constructor**.
- The newly created variable will be immediately installed in existing modeling objects.

Modeling by Column

- For example:

  IloNumVar var = cplex.numVar(col, 0.0, 1.0);

  – This creates a new variable with bounds 0.0 and 1.0 and immediately installs it in the objective obj with linear coefficient 1.0 and in the ranged constraint rng with linear coefficient 2.

- Methods for constructing arrays of variables take an IloColumnArray object as a parameter that defines how each individual new variable is to be installed in existing modeling objects.

buildModelByColumn Method

- First, the method creates an empty minimization objective and empty ranged constraints, and adds them to the active model.

```
IloObjective cost = model.addMinimize();
IloRange[] constraint = new IloRange[nNutrs];
for (int i = 0; i < nNutrs; i++)
    {
      constraint[i] =
            model.addRange(data.nutrMin[i], data.nutrMax[i]);
    }
```

- Empty means that they use a 0 expression.

buildModelByColumn Method

- After that the variables are created one by one, and installed in the objective and constraints modeling by column.
- For each variable, a column object must be created.
- Start by creating a column object for the objective by calling:

  ```
  IloColumn col = model.column(cost, data.foodCost[j]);
  ```

- The column is then expanded to include the coefficients for all the constraints using col.and with the column objects that are created for each constraint, as in the following loop:

```
for (int i = 0; i < nNutrs; i++) {
    col =
    col.and(model.column(constraint[i], data.nutrPerFood[i][j]));
}
```

buildModelByColumn Method

- When the full column object has been constructed it is finally used to create and install the new variable like this:

```
Buy[j] =
  model.numVar(col, data.foodMin[j], data.foodMax[j], type);
```

- The diet program:
  - Diet.java

References

References

- ILOG CPLEX, **ILOG CPLEX User's Manual,** ILOG CPLEX, 2008.
- ILOG CPLEX, **ILOG CPLEX Java API Reference Manual,** ILOG CPLEX, 2008.

The End