Optimization modeling with IBM ILOG OPL

Instructor workbook

IBM ILOG OPL
Course version: 6.3
Course reference: OPL63-tr1_0731 / Document ID: OPL63-WB-IN-tr1_0731

Contents

About this course

This section provides you with a brief description of the course, audience, suggested prerequisites, and course objectives.

Course description

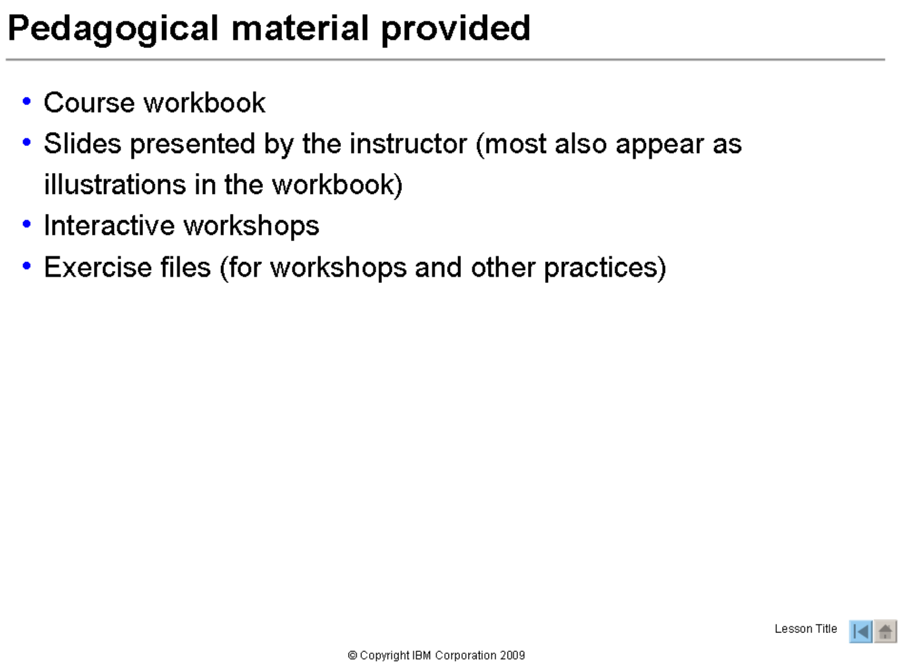
This is a 3-day course on the fundamentals of IBM® ILOG® Optimization Programming Language (OPL) Development Studio.

Audience

Specialists who need to design models to solve business optimization problems.

Course objectives

- Learn to use IBM ILOG OPL by solving practical problems
- Gain a working knowledge of OPL features and functions
- Understand the role of OPL and its relationship to other tools in the solving of complex business optimization problems

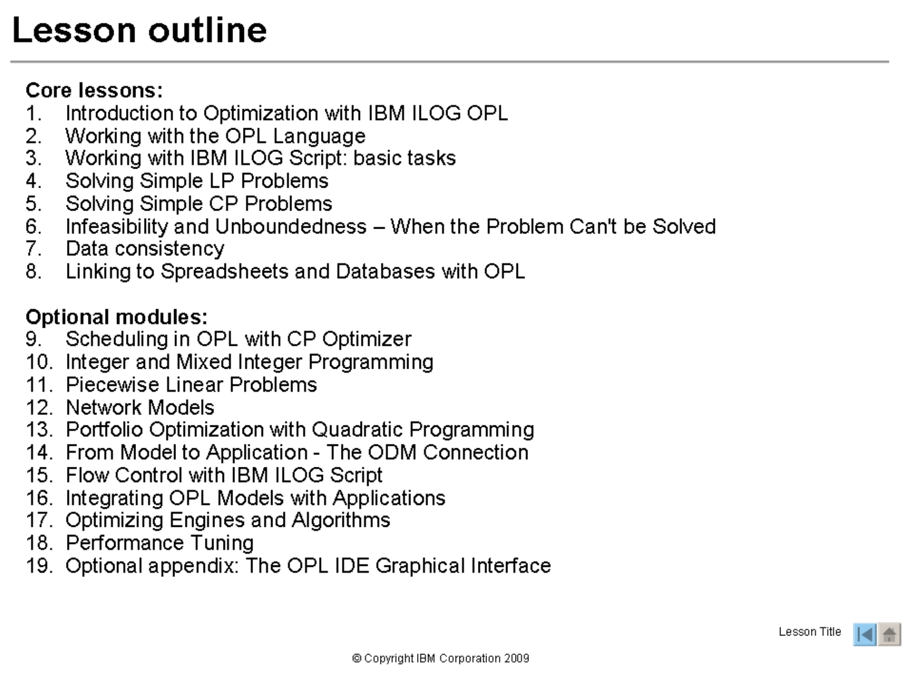


Pedagogical material provided

- Course workbook
- Slides presented by the instructor (most also appear as illustrations in the workbook)
- Interactive workshops
- Exercise files (for workshops and other practices)

Lesson Title

© Copyright IBM Corporation 2009



Lesson outline

Core lessons:

1. Introduction to Optimization with IBM ILOG OPL
2. Working with the OPL Language
3. Working with IBM ILOG Script: basic tasks
4. Solving Simple LP Problems
5. Solving Simple CP Problems
6. Infeasibility and Unboundedness – When the Problem Can't be Solved
7. Data consistency
8. Linking to Spreadsheets and Databases with OPL

Optional modules:

9. Scheduling in OPL with CP Optimizer
10. Integer and Mixed Integer Programming
11. Piecewise Linear Problems
12. Network Models
13. Portfolio Optimization with Quadratic Programming
14. From Model to Application - The ODM Connection
15. Flow Control with IBM ILOG Script
16. Integrating OPL Models with Applications
17. Optimizing Engines and Algorithms
18. Performance Tuning
19. Optional appendix: The OPL IDE Graphical Interface

Lesson Title

Prerequisites

- Working knowledge of the Microsoft Windows operating system
- Knowledge of basic algebra
- For problems that use CPLEX, basic knowledge of mathematical programming and/or modeling concepts

How to use

This workbook provides you with the information that you need in order to present the main concepts of OPL to the students. The slides that accompany the workbook show the structure of the workbook (through its lesson and topic headings) as well as some of the graphics and main points.

Use the slides as a guide, and to prompt you to cover topics in the correct order, but you will need to refer to the workbook to prepare the complete standard content.

The material is divided into three groups of modular lessons:

- Core lessons – the basic information needed to use OPL
- Additional modules – additional topics of interest to many users

See the HTML **Notes for Instructor** page for details of how these are organized. You will also find a guide as to how you can present an MP-only or CP-only progression.

You should go through the core material in sequence as it is presented in the workbook, adding examples and explanation as appropriate. It is possible to insert some optional topics between the core lessons if it seems appropriate.

Additional modules should be programmed by you as a function of expressed needs of the customer and interests of your training group.

There are instructor notes throughout the workbook that are only available in your copy of the workbook. These either remind you of particular points to make that are not part of the main flow of the workbook, or contain additional information and hints to help you illustrate a point.

Some lessons contain practices. Practices can be any of the following:

- Labs (indicated by a small computer icon) are formal exercises accessed from your web browser that have associated OPL model and data files. They are to be performed by the students with your guidance.
- Hands-on (indicated with a small hand icon) are less formal demonstrations and exercises. They can be carried out by both the instructor and the students, or just used by the instructor to demonstrate a particular feature. Students should try to complete all the hands-on practices to further their understanding of the product.
- Discussions (indicated with a small icon of two people talking) are informal and can be used to start a discussion or exploration of a theme developed during the training sessions.

`<TrainingDir>`refers to the directory in which the practical part of the training is stored. Usually this is a directory that can be found directly on C:, for example: **`C:\OPLTraining`**. This contains a directory of lab files, including both work and solution files.

You should encourage the trainees to consult the user documentation, as it contains descriptions of procedures that may help them to complete the hands-on practices. The documentation can be found within the standard help system of the graphical environment provided in the OPL IDE. To access the OPL Online Help, click **Help** on the **Help** menu. Dynamic contextual help for many functions in OPL and IBM ILOG Script is available via the **`F1`** key.

Typographical conventions used in this workbook

Typography	Meaning	Example
`Text in fixed width font`	Code sample, screen output, directory paths or file names, user input to be typed	**`forall (p in products) production[p] >= Min[p];`**
`<Fixed width font in angle brackets>`	Token for data (in code samples or templates)	**`<FileName>`**
	XML tag (in code samples or templates)	**`<Explanation> </Explanation>`**
	Key press other than a character	**`<enter>`**
Bold text	Menu selections, buttons, navigation tree items	**File >New >New Default Scenario**
	Key terms in the body of the workbook	In ODM, **scenarios** are grouped together in one or more **workspaces**.

Typography	Meaning	Example
`|` (vertical bar)	In a model file, represents the mathematical **such that**	`int a[1..10] = [ i-1 : i | i in 2..11 ];`
	In a command line or syntax model, exclusive **or**	`<type> <name>[index] = <member1>, <member2>; | ...;`
The use of `{curly braces}` or `[square brackets]` is as per the language sampled (e.g. Java™, OPL language, XML, etc.)		
Parentheses – `( )` – in code samples are used only when required by the syntax, and are thus to be typed in all languages		

Classroom information

Read through the workbook paying particular attention to the instructor notes.

Classroom preparation list

- Try all demonstrations/practices to make sure that you are familiar with their aims, and that OPL [ODM] and the training material are correctly installed.
- Read through the slides making sure that you are familiar with the main points in the sections that they reference.
- Verify the installation of OPL and the training practice files on each machine in the classroom.
- Try all the labs and hands-on practices and make sure you are able to complete them without problems.

Classroom setup list

- This list identifies what must be present in the classroom
- Projector
- Whiteboard
- Up to ten PCs each with the following installed:
    - Any one of the following: Microsoft® Windows® Vista, Windows XP Professional, Microsoft Windows 2003 Server
    - Microsoft Excel 2003 Professional or higher
    - Microsoft Access 2003 Professional or higher
    - OPL V6.3
    - OPL V6.3 training material
    - ODM V3.3 (if you intend to demonstrate the ODM connection in your training)
    - To do the API based labs, you will need one or more development environments such as JDK V5.0 or higher, Eclipse or Microsoft Visual Studio.

To start the class, introduce yourself to the students and have them introduce themselves to the group. You may also want to discuss logistical details: coffee breaks, lunch, and so on.

A note about the timings of each lesson. These are given at the beginning of each lesson in the instructor's workbook only, and are very approximate.
It is impossible to deliver all the modules of this training in three days. You should determine which of the optional lessons are important for your trainees, and organize the material in a logical sequence. In general, all the core lessons should be presented (you can skip Lesson 3 for CP-only or lesson 4 for MP-only). Feel free to insert some optional lessons in between the core lessons if it suits you for timing or pedagogical purposes.

Depending on local customs about lunch hour, etc. you may need to break a single lesson in the middle. If so, plan ahead where that break should occur so that students can keep continuity before and after the break.

Lesson 1: Introduction to Optimization with IBM ILOG OPL

Instructor note
This lesson should last about 1 hour

In this lesson, you will meet IBM® ILOG® OPL and see how it fits into the IBM ILOG Optimization Suite of products. You'll learn about the OPL Integrated Development Environment (IDE) and the powerful IBM ILOG optimization engines available through OPL. You'll get an overview of the functionality available through the OPL IDE, and understand how OPL projects are managed. In addition, you'll learn how to recognize the basic optimization model elements when looking at an OPL model, and understand what IBM ILOG Script is.

By the end of this lesson, you will:

- Be able to understand how the IBM ILOG Optimization Suite components work together to solve complex problems, and OPL's role in it.
- Know the optimization techniques and be able to identify the optimization engines available through OPL.
- Understand the basic functionality of the OPL IDE.
- Understand what an OPL project is and how OPL projects are organized and managed.
- Know how to recognize the basic model elements, namely data, decision variables, objectives and constraints, when looking at an OPL model.

The big picture: IBM ILOG Optimization Suite

Learning objective
Understand the interworking of the IBM® ILOG® Optimization Suite components as a complete solution system

Key terms
- IBM ILOG Optimization Suite
- OPL
- ODM
- application development and deployment

The **IBM ILOG Optimization Suite** is a set of products for developing custom optimization applications that use IBM's powerful optimization engines.

It provides a complete support system for the optimization application development process. Modelers and IT personnel can collaborate to rapidly develop and deploy state-of-the-art planning, scheduling and other optimization applications.

The components of the IBM ILOG Optimization Suite

- Model development tool: IBM ILOG OPL
- Application development tool: IBM ILOG Optimization Decision Manager (ODM)
- Optimization engines:
    - IBM® ILOG® CPLEX® for Mathematical Programming (MP)
    - IBM ILOG CP Optimizer for Constraint Programming (CP)

What is IBM ILOG OPL?

IBM ILOG OPL is an integrated development environment (IDE) for optimization model building, debugging and tuning.

- OPL models are written with easy-to-use declarative language.
- OPL is designed to take maximum advantage of IBM ILOG CPLEX and IBM ILOG CP Optimizer:
- An OPL model can be used standalone, or as part of an ODM application.
- OPL includes a scripting language (IBM ILOG Script) for pre- and post-processing, as well as flow control.

What is IBM ILOG ODM?

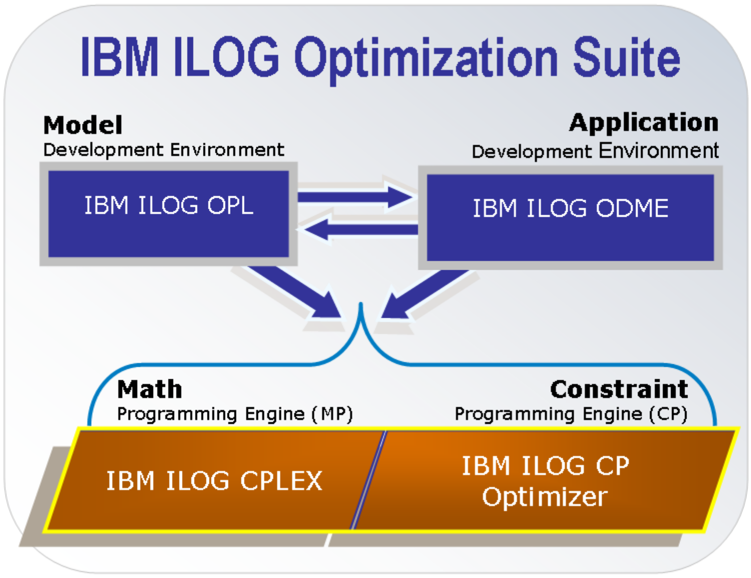
IBM ILOG ODM is a specialized application development and deployment tool used to build highly interactive, state-of-the-art, custom decision support applications.

- ODM facilitates collaboration between
    - Operations Research (OR) experts,
    - Information Technology (IT) professionals, and
    - Business decision makers.
- ODM includes advanced functionality such as scenario creation and comparison, what-if analysis, and powerful graphics and charting tools.
- An ODM application is generated through the OPL IDE.
- An ODM application can use an OPL model, or any other optimization model called through the OPL IDE (for example, a model written in Java™ and called in the OPL IDE using IBM ILOG Script).

To sum up:
- Use OPL for modeling
- Use ODM for prototyping, application development and deployment

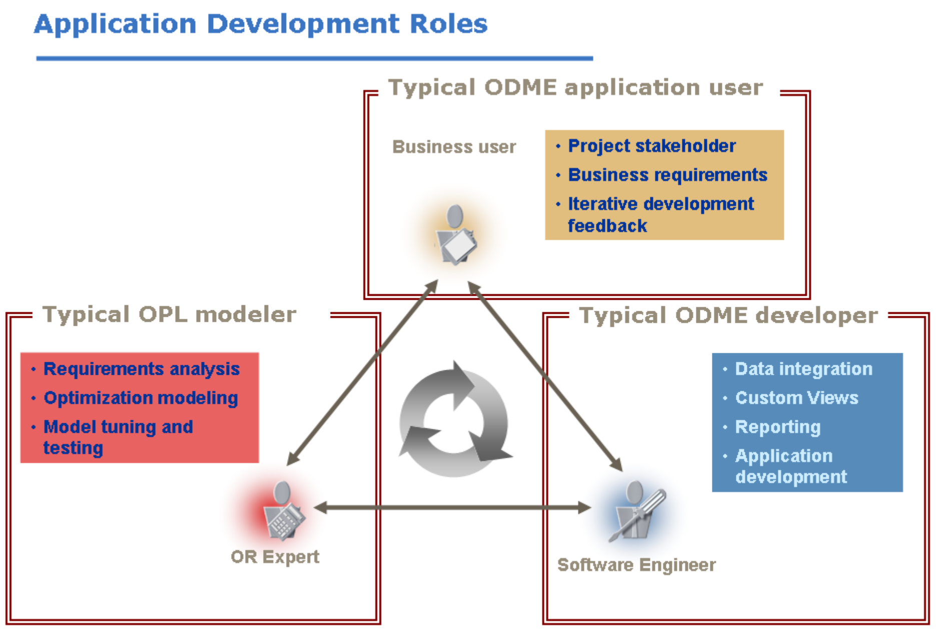
as per the following diagram:

IBM ILOG Optimization Suite Interactions



Roles and goals

The IBM ILOG Optimization Suite is designed to facilitate the work of different people involved in an application powered by optimization.



OPL is a tool for building optimization models. It is typically used by OR specialists who work with other experts, for example:

- Software experts who perform data integration and integration with the target deployment environment.
- Decision makers who provide the business and application requirements, and who may also be the end users.

ODM is both an application environment, used by business users to do their analysis, and an application development and deployment tool, used by OR and IT professionals to develop custom applications.

The goals of the IBM ILOG Optimization Suite are:

- Allow creation of intuitive, interactive optimization applications:
    - Provide what-if analysis tools
    - Furnish understandable solutions to complex schedules and plans
    - Leverage the power of optimization specialists
    - Leverage the insight of humans where the plan or schedule meets reality
- Reduce development time and risk via tight integration of different components of the system
- Allow rapid prototyping of applications
- Encourage and augment business stakeholders' participation in the model refinement and application development processes

Inside OPL

Learning objective
Get an overview of the capabilities and features of OPL.

Key terms
- OPL IDE
- optimization model
- optimization engine
- OPL project

You will now take a brief tour of OPL and discover its capabilities and features.

OPL in a nutshell

OPL = Optimization Programming Language

Optimization modelers use OPL to create and test optimization models consisting of a combination of data, decision variables, objectives and constraints.

OPL is available as a standalone product or through ODM Enterprise as the OPL perspective in the ODM Enterprise IDE.

Optimization techniques in OPL

The optimization techniques available in OPL are:

- Mathematical Programming (MP), specifically
  - Linear Programming (LP)
  - Integer Programming (IP)
    In practice, many models that require integer decision variables also require some continuous decision variables, and so it is more common to use Mixed Integer Programming (MIP) than pure IP.
  - Quadratic Programming (QP)
- Constraint Programming (CP), typically used for
  - Detailed scheduling
  - Certain combinatorial problems not well-suited for MP

In the context of MP, the word “programming” does not necessarily mean computer programming, but refers to a problem-solving methodology. The origins of this term come from the development of military “programs” to which the technique was originally applied. In the CP context, unlike MP, the word “programming” refers to a computer programming methodology.

Instructor note
Note that OPL releases V4.0–V5.1 do not support CP. CP support with a new engine, IBM ILOG CP Optimizer, was reintroduced with release V5.2. OPL V5.2 does not support IBM ILOG CP.
Students who have used an earlier version of OPL to write scheduling applications based on constraint programming need to migrate their OPL V3.7 (or earlier) projects to OPL V6.3. Instructions for doing this can be found in the documentation.

In the case of MP problems, OPL can represent the following types of objective functions, data, decision variables, and constraints:

When the objective function is...	Decision variables and data elements can be...	And constraints can be...
Linear	• Integer • Continuous	• Linear inequalities (<= , >= , ==) • Logical relations (&&, | |, !, !=)
Quadratic	• Integer • Continuous	• Linear or quadratic inequalities (<= , >= , ==) • Logical relations (&&, | |, !, !=)

Not all quadratic problems can be addressed with OPL.

It is possible to have an MP model without an objective function. In this case, OPL will simply instantiate data in the model without solving it.

In the case of CP, OPL can represent the following types of objective functions, data, decision variables, and constraints:

The objective function...	Decision variables can be...	Data elements can be...	And constraints can contain...
may be present or absent	• integers with defined domain • intervals (for scheduling)	• integer • real	• arithmetic operations, expressions and constraints • logical relations (and, or, not, if-then) • allowed and forbidden assignments • specialized constraints (allDifferent, allMinDistance, inverse, lex, pack)

Optimization engines

IBM® ILOG® OPL gives the user access to the following solver engines:

- IBM® ILOG® CPLEX® for **Mathematical Programming (MP)**
- IBM ILOG CP Optimizer for **Constraint Programming (CP)**

OPL's default setting is to use CPLEX.

The components of OPL

OPL consists of the following components:

- The OPL IDE (Windows® 64 and 32 bits) in which you write, execute, test, and debug optimization models. In ODM Enterprise, this is the OPL perspective of the ODM Enterprise IDE.
- IBM ILOG Optimization Programming Language (OPL), which allows you to write optimization models in a declarative way
- IBM ILOG Script, a scripting language for OPL
- Application Programming Interfaces (APIs) to embed models into standalone applications

- The ODM connection to automatically generate end-user applications (requires a license for ODM when using OPL as a standalone product)

Writing OPL models in a declarative way means that you write them in a similar way as you would on paper, except that you use OPL syntax.

OPL also includes **`oplrun,`** a tool to launch OPL from the command line.

OPL is built on IBM ILOG Concert Technology, which links with the IBM ILOG CPLEX and IBM ILOG CP Optimizer optimization engines. Complete access to engine algorithmic settings is provided.

When using OPL to generate ODM applications, you do not work with the **`oplrun`** command, and instead generate the application directly from the IDE.

The OPL IDE

The OPL IDE (or the OPL perspective in the ODM Enterprise IDE) includes:

- An editor to create OPL models
- Views of data and solutions
- Debugging capabilities
- Online help, including contextual help
- Various other views to facilitate analyzing your model and solution

Instructor note
There are plans for the OPL IDE to also be available for Linux and/or MaxOS in the future, but no specific date is set yet.

The OPL language

This high-level language provides:

- A compact declarative language for Mathematical Programming (MP) and Constraint Programming (CP)
- Advanced data types
- Connections to relational databases and Excel spreadsheets
- The ability to call external Java™ functions from inside OPL
- IBM ILOG Script

IBM ILOG Script

This scripting language is used for:

- Preprocessing of data and engine parameters
- Postprocessing of solutions, or data. or both
- Flow control, for example, in decomposition or incremental modification of the model

The OPL APIs

The APIs allow you to embed OPL models and scripts into applications using:

- C++
- Microsoft® .NET
    - Visual Basic.NET, C#, etc.
    - Microsoft Office 2003 or higher via Visual Studio Tools for Office
- Java™
- Web
- ASP.NET, JSP

The ODM connection

If you have the desktop version of ODM installed on your system together with the standalone version of OPL, the OPL-ODM connection allows generation of an ODM application from the OPL IDE. An ODM application can be based on one or more OPL models.

Instructor note

The ODM connection runs on Windows® installations only.

Models, data and projects

OPL lets you construct **models** which are independent of the **data** they use. This means that you don't need to change your model each time the data changes. The model is maintained as a separate entity (model file) from the data (data file), and is therefore reusable for multiple instances of the same problem with different input data.

One or more model files and, optionally, one or more data files are grouped together into a **project,** which, in addition to models and data, contains control information to instantiate one or more problems.

The project control information is contained in 2 types of structure:

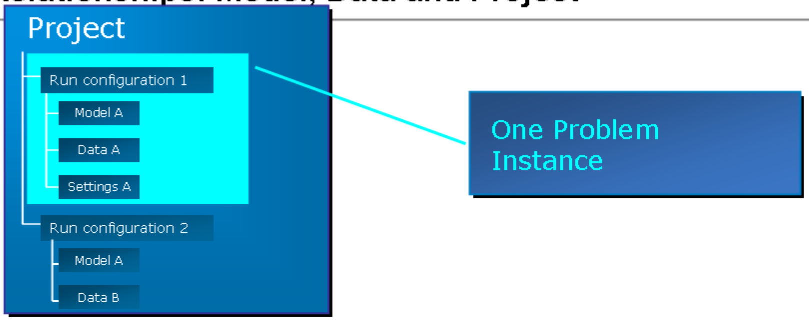
- One or more settings files (`.ops` files) that control such elements as conflict resolution, choice and behavior of linear programming algorithms, and constraint programming control parameters.

  If a settings file is not specified, OPL will use the default settings. Settings can also be specified using IBM ILOG Script. A settings file can be added to a project at any time.

- One or more run configurations. These are assemblies of models, data and settings files that are meant to run together.

The relationships between these files are shown in the following diagram.

Relationships: Model, Data and Project



A model is a mathematical representation of a problem.
In OPL, models can exist independent of data.

A project consists of one or more models, and, optionally, one or more data files and a settings file.

A run configuration specifies a problem instance by linking the models, data, and settings.

Lesson Title

More about OPL projects

OPL projects are located inside directories, typically with the same name as the project, on the computer's file system. These directories typically contain:

- The project description files (**`.project`** and **`.oplproject`**)
- Model files (**`.mod`**)
- Data files (**`.dat`**)
- Settings files (**`.ops`**)

The project files are XML files that contain a technical description of the project.

Model files contain data declarations, as well as the model definition in terms of decision variables, the objective function and constraints. Model files may also contain IBM ILOG Script statements.

Data files initialize the data declared in the model files. Data can be initialized directly in the **`.dat`** file, or imported from external sources.

Setting files are used to change the default settings in OPL, for example parameters that define solution algorithm behavior, display options, and so forth.

The directory for a particular project serves as a container for all the related files and control information associated with the project. It provides a convenient way to group all the related model, data and settings files for the project, and also maintains information about the relationship between files and runtime options for the environment.

More about run configurations

- Run configurations represent different combinations of model, data and settings files associated with the same project.

- They are defined within an OPL project in the OPL IDE (or OPL perspective in the case of ODM Enterprise).
- A run configuration includes at least one model file.
- A run configuration can include multiple model and data files, and at most one settings file.
- You can define as many run configurations as you need within a given project.

It is possible to use more than one settings file in a project, and to attach each to a different run configuration. This is very useful to, for example, test different algorithmic settings on the same model.

A minimal project has:

- One model file
- One default run configuration referencing that same model file

A typical project has:

- One or more model files
- Any number of data files (or no data file)
- One or more settings files
- One or more run configurations referencing various combinations of the model, data, and settings files

Standalone models (i.e. model files that are not attached to a project) are not supported in the IDE. It is, however, possible to use standalone models with the **`oplrun`** command.

A quick look at the OPL IDE

You instructor will now guide you in taking a quick look at the OPL IDE (or the OPL perspective in the case of ODM Enterprise)

Instructor note

Spend about 15 minutes on this demo. Open a project in the OPL IDE or the OPL perspective in the case of ODM Enterprise, and point out the following:

- Editing area
- Model outline
- Main toolbar
- OPL projects navigator
- Problem browser
- Output tabs
- Status bar

Example: a production planning problem

Learning objective
Gain familiarity with the OPL IDE. Learn how to recognize data, decision variables, objectives, and constraints when looking at an OPL model.

Key terms
- OPL IDE
- OPL project
- run configuration
- data file
- model file
- decision variable
- objective function
- constraint

In the example that follows, you'll gain some familiarity with the OPL IDE and how projects and run configurations are structured. You'll also learn to recognize the following when looking at an OPL model:

- Data declarations
- Decision variables
- Objective functions
- Constraints

Problem description

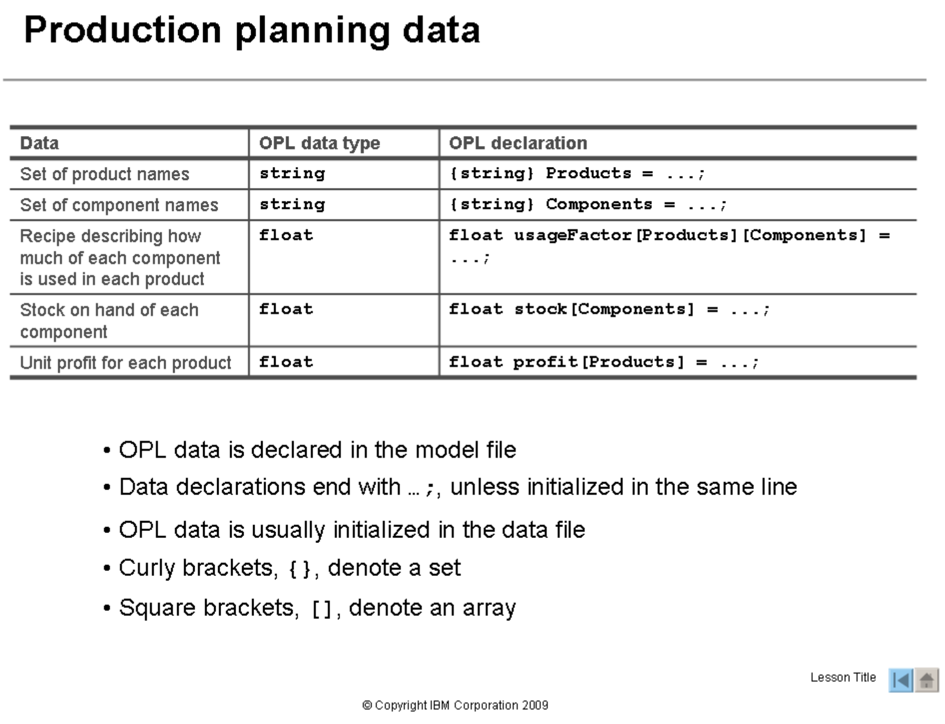
Consider a typical production planning problem where a company produces a number of products. Each product has a unit profit associated with it, and is made up of different components. The company's objective is to maximize the profit while using only the available stock of components.

In the next steps you'll see how this problem is written using OPL syntax in terms of the data, decision variables, objective and constraints.

Here, you will concentrate on a simple example. A more complete explanation of OPL syntax follows in a later lesson.

The data

The following table shows the data for this problem, together with the OPL data type and OPL data declaration:



Production planning data

Data	OPL data type	OPL declaration
Set of product names	`string`	`{string} Products = ...;`
Set of component names	`string`	`{string} Components = ...;`
Recipe describing how much of each component is used in each product	`float`	`float usageFactor[Products][Components] = ...;`
Stock on hand of each component	`float`	`float stock[Components] = ...;`
Unit profit for each product	`float`	`float profit[Products] = ...;`

- OPL data is declared in the model file
- Data declarations end with `...;`, unless initialized in the same line
- OPL data is usually initialized in the data file
- Curly brackets, `{}`, denote a set
- Square brackets, `[]`, denote an array

Lesson Title

© Copyright IBM Corporation 2009

The decision variables

For the production problem, the decision variables are the quantities to produce of each product in the set **`Products`**. This can be written as follows using OPL syntax:

```
dvar float+ production[Products];
```

Here, **`dvar`** is the OPL keyword used to declare decision variables.
`float` is the OPL keyword used for real numbers, and the **`+`** is added to denote that the quantities are non-negative.
`production` is our choice of variable name, and it is defined as an array over the set of **`Products`**.

Note that all declarations in OPL end with a semicolon.

The objective function

The objective is to maximize the total profit over all **`Products`**, where the profit for each product, **`p,`** is defined as the unit profit, **`profit[p],`** multiplied by the quantity produced, **`production[p]`**. The objective can be written as follows using OPL syntax:

```
maximize sum(p in Products) profit[p] * production[p];
```

Here, **`maximize`** is the OPL keyword used to declare an objective to be maximized.
`sum` is the OPL keyword to compute the summation of a collection of expressions, in this case to sum up the profit over all products. Note that we use normal parentheses, as in **`(p in Products)`**, to denote the selection to sum over.
`p` is an index used to access each element of the set of **`Products`**.

The constraints

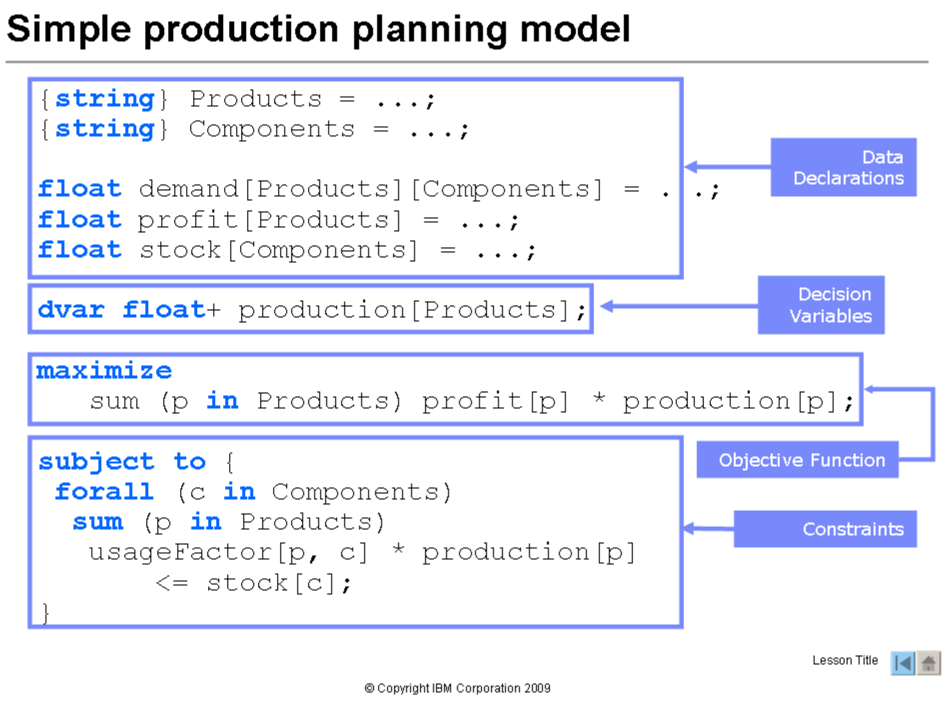
For this problem, the constraints are that, for each component, the amount of component used across all products (according to the **`usageFactor`** of that product) should not exceeded the available stock. This can be written as follows using OPL syntax:

```
subject to{
 forall(c in Components)
  sum(p in Products) usageFactor[p,c] * production[p] <= stock[c];}
```

In the OPL IDE, constraints are written inside a block starting with **`subject to {`**, and ending with **`}`**
Here, **`forall`** is the OPL keyword used when expressions are similar, except for their indices. In this case, it's used to write only one constraint for all components, seeing that the constraints only differ according to the product or component they refer to.
`c` is an index used to access each element of the set of **`Components`**.

Complete problem formulation

The following diagram shows how the complete problem would look in the OPL IDE.



Simple production planning model

```
{string} Products = ...;
{string} Components = ...;

float demand[Products][Components] = ...;
float profit[Products] = ...;
float stock[Components] = ...;

dvar float+ production[Products];

maximize
   sum (p in Products) profit[p] * production[p];

subject to {
 forall (c in Components)
  sum (p in Products)
   usageFactor[p, c] * production[p]
       <= stock[c];
}
```

© Copyright IBM Corporation 2009

A data instance

To be able to solve the model, it first has to be populated with data. Suppose that the products to be produced are chemicals. The following table shows example data that can be used in the model, and how it is can be instantiated in the OPL IDE (hard-coded data instantiation is usually done in the .dat file):

Data Description	Instantiation using OPL syntax
Products are: • Ammonium gas (NH3) • Ammonium chloride (NH4Cl)	`Products = { "gas", "chloride" };`
Components are: • Nitrogen (N) • Hydrogen (H) • Chlorine (Cl)	`Components = { "nitrogen", "hydrogen", "chlorine" };`
Usage of components is: • 1 unit of nitrogen and 3 units of hydrogen to produce 1 unit of gas • 1 unit of nitrogen, 4 units of hydrogen, and 1 unit of chlorine to make 1 unit of chloride	`usageFactor = [ [1, 3, 0], [1, 4, 1] ];`
Stock on hand is: • 50 units of nitrogen • 180 units of hydrogen • 40 units of chlorine	`stock = [ 50, 180, 40 ];`

Data Description	Instantiation using OPL syntax
Profit for each product is: • gas = 30 • chloride = 40	`profit = [ 30, 40 ];`

In a real application you will not normally hard code data in the data or model files, but rather link to external databases or spreadsheets. OPL's spreadsheet and database linking abilities will be discussed in another lesson.

Practice

Familiarize yourself with the OPL IDE
In this practice, you'll get familiar with the OPL IDE using the gas production problem. To start, launch the OPL IDE by clicking its icon on your desktop or in the Windows start menu (**Start > All Programs > IBM ILOG > OPL**).

Steps:

1. Import the project for this problem by selecting **File > Import > Existing OPL 6.x projects** from the main menu, navigating to the **`<TrainingDir>\OPL63.labs\Gas\work\gas`** directory in the **Select root directory** field, and selecting the listed project. Click **Finish**.
2. Expand all the plus signs in the OPL Projects Navigator to see that the project contains two files: **`gas.mod`** (the model file) and **`gas.dat`** (the data file), as well as a default run configuration (labeled **Default (default)**). Both the model and data files have been associated with this run configuration.
3. Double-click the model or data file names to look at the contents together with your instructor.

   Note that in the OPL IDE, keywords are highlighted in blue, comments in green, and string data in purple.

4. Try using contextual help by first selecting **Help > Dynamic Help** from the main menu to open the help window on the right side of the OPL IDE, and then highlighting any keyword in the model file that you'd like information on.
5. Run the project by right-clicking the default run configuration in the OPL Projects Navigator, and selecting **Run this** from the context menu. Examine the result and different output tabs together with your instructor.

Debugging

The OPL IDE provides debugging facilities to trap errors such as syntax and runtime errors.

Errors are listed in the **Problems** Output tab, and are also indicated with an icon in the text editor.

Advanced debugging features, such as breakpoints in IBM ILOG Script blocks, are available – for more information see the Appendix and the OPL online help.

Practice

Debugging an error message
In this practice, you'll see how to debug an error message for the gas model in the OPL IDE.

Steps:

1. Introduce a syntax error in the **`gas.mod`** file by removing the semicolon (;) from the end of one line.
2. Observe the message that appears in the **Problems** Output tab (near the bottom of the screen) and the mark in the margin of the text editor.
3. Try to run the default run configuration (accept the option to save) and see in the **Problems** Output tab that this is not possible before resolving the error.
4. Fix the syntax error, save the model file, and run it.

Instructor note
Here, as a function of the students' knowledge level, you can, if you wish, go into more detail about debugging. For example, you can demonstrate breakpoints using a more complex model. The file **`mulprod_main`**, supplied in the examples, contains a script with a loop. Introduce a breakpoint at the command **`best = cur;`** (line 61) and run the model to demonstrate this feature. Again, depending on the level of your group, you might want to defer this demonstration to Lesson 13, which deals with IBM ILOG OPL Script. If, on the other hand, you think they don't need it, skip this demonstration altogether.

Practice

Model and data independence
We've mentioned before that the model and data are usually independent entities in the OPL IDE. In this practice, you'll see how you can use the production model from the gas project, and combine it with different data to create a new project for a jewelry production problem.

The data for jewelry production is as follows:

Instructor note
You should orient this practice to the knowledge level and job functions of the audience. While the primary objective of the practice is to demonstrate the use of different data files with a single model, this demo will also introduce the students to the user interface in a general way. Be prepared to show them where commands are found, and the alternative ways of accessing them (menu bar, toolbar buttons, keyboard shortcuts).

It will be obvious to an experienced user that simply adding the **`jewelry.dat`** file and a new run configuration to the **`gas.prj`** project would suffice for this example. Run configurations will be dealt with in a later practice, so for the moment, we create two separate projects. You should feel free to explain, however, this shorter method if you feel it will help at this time.

Data Description	Declaration in OPL
Products are: • Rings • Earrings	`Products = { "rings", "earrings" };`
Components are: • Gold • Diamonds	`Components = { "Gold", "Diamonds" };`
Usage for components is: • 3 units of gold and 1 diamond to produce 1 ring • 2 units of gold and 2 diamonds to make 1 set of earrings	`usageFactor = [ [3, 1], [2, 2] ];`
Stock on hand is: • 150 units of gold • 180 diamonds	`stock = [ 150, 180 ];`
Profit for each product is: • ring = 60 • earrings = 40	`profit = [ 60, 40 ];`

Write a new OPL data file to use with the existing production model.

Steps:

1. Create a new project in the OPL IDE by selecting, from the menu bar, **File> New> OPL Project.** When the **New Project** dialog box opens, enter **`jewelryWork`** as a **projectName**.

2. Use the **Browse** button next to the **Location** field to navigate to **`<TrainingDir>\OPL63.labs\Gas\Work`** and make a new folder called **`jewelryWork`**. You will have created the **`<TrainingDir>\OPL63.labs\Gas\Work\jewelryWork`** directory. Click the **Finish** button. The project opens in the IDE with a blank model file.
3. In the **Projects** window, right-click **`jewelryWork.mod`** and select Delete from the context menu. Instead of creating a model file from scratch, you are going to use an existing model: the **`gas.mod`** model.
4. In the OPL Projects Navigator select the **`jewelryWork`** project. From the menu bar, select **File > Copy Files From Project**. Browse to **`<TrainingDir>\OPL63.labs\Gas\Work\gas`** through the **From Directory** field, and select the **`gas.mod`** file only. You can then click **Finish**. The **`gas.mod`** file is added to the current project
5. In the OPL Projects Navigator, drag the **`gas.mod`** file into **Configuration1**
6. Use the same procedure to copy the **`jewelry.dat`** data file from **`<TrainingDir>\OPL63.labs\Gas\solution\jewelrySolution`** into the project and add it to **Configuration1**
7. Run the project. Do you get a valid result?
8. With your instructor, compare the result with the result from **`<TrainingDir>\OPL63.labs\Gas\solution\jewelrySolution`** and with the result of running **`<TrainingDir>\OPL63.labs\Gas\work\gas`**

Summary

Review

In this lesson, you learned how OPL and IBM ILOG optimization technology can help you make business decisions.

IBM ILOG OPL is a component of the IBM ILOG Optimization Suite, a set of tools that also includes IBM ILOG Optimization Decision Manager (ODM) and the IBM ILOG optimization engines, IBM ILOG CPLEX and IBM ILOG CP Optimizer. IBM ILOG Optimization Suite is used to develop models and applications based on Mathematical Programming (MP) or Constraint Programming (CP).

OPL allows development of custom solutions to business optimization problems, using a low-level declarative language. The OPL IDE is an intuitive interface that includes features such as a text editor for model development, data and solution views, debugging features, online help, and IBM ILOG Script (a scripting language for pre- and postprocessing and flow control).

OPL facilitates model and data independence. OPL models can either use data declared in OPL data files, or data from external sources such as databases or excel files.

An OPL project is a collection of OPL model files, data files, and setting files, that can be grouped into various run configurations. An OPL project can usually be found in a directory with the same name on your computer's file system. The contents of that directory are files containing the project description, models, data, and settings.

OPL models can be integrated into applications developed with IBM ILOG Optimization Decision Manager (ODM), allowing business users to perform what-if analysis, and compare scenarios. OPL models can also be integrated into external applications using the OPL APIs.

In this lesson, you've gained familiarity with the OPL IDE and seen how a simple project looks. You've seen an example of how data, decision variables, objectives and constraints are declared in an OPL model, and how data can be instantiated in an OPL data file. You've also practiced some simple debugging, as well as creating a new project using two separate model and data files.

Lesson 2: Working with the OPL Language

In the lesson introducing OPL, you looked at a simple production planning model in OPL, and saw the model structure including the data declarations, decision variables, objective and constraints. In the current topic, you will gain a deeper understanding of these model elements, and be introduced to some of the other elements that can be present in an OPL model.

OPL is a declarative language. This means that you do not need to write procedures when constructing a model. Instead, you simply declare your data elements, decision variables, objectives and constraints, then let OPL call a solver engine to solve the model for you. You can optionally add some procedures for pre- and postprocessing, as well as flow control, using IBM® ILOG® Script.

In this lesson, you will learn how to write a model in OPL using OPL syntax, together with the available OPL data structures and operators, while basic IBM ILOG Script functionality is covered in the next lesson.

At the end of this lesson you will be able to:

- Describe the structure of an OPL model
- Describe the data types, data structures, and types of variables available in OPL
- Describe some of the constraints available in OPL
- Understand the concept of sparsity
- Write a simple model using OPL syntax

Instructor note
This is, perhaps, the most fundamental lesson of the entire training course. You will need 2.5 – 3 hours for it, including the practice. If your students master this lesson, the rest of the course will be much easier to explain.

OPL model structure

Learning objective
Learn how an OPL model is structured.

Key terms
- data
- decision variable
- objective
- constraints
- IBM ILOG Script

An OPL model is typically structured in the following sequence (some of these are optional, as you'll learn in this topic):

- The choice of solver engine
- Data declarations
- Decision variables
- Objective function
- Constraints

An OPL model file can also contain IBM® ILOG® Script statements:

- before the objective for preprocessing
- after the constraints for postprocessing
- before the objective or after the constraints for flow control

From OPL 5.0 onwards, it is illegal to insert any statements between the objective function and the constraints.

This lesson focuses on the OPL language only, and does not cover IBM ILOG Script.

The choice of solver engine

The following two solver engines are available in OPL:

- CPLEX® for Mathematical Programming (MP) problems.
- CP Optimizer for Constraint Programming (CP) problems, specifically:
  - Detailed scheduling problems
  - Certain combinatorial problems not well-suited for MP

OPL uses CPLEX by default. To specify that CP Optimizer should be used, start your model with the following text:

```
using CP;
```

You can also explicitly state that CPLEX should be used by starting the model with **`using CPLEX;`**. If you use constraint-programming keywords in your model but do not specify CP as the solver engine, OPL will return syntax errors. Also, the types of decision variables and constraints are dependent on the choice of solution engine, and it is therefore important to make this choice before starting to construct your model.

If you're not that familiar with MP or CP, an expert in either field will be able to tell you which engine to use based on your problem description.

Data

When declaring data, you need to decide:

- The name for the data item
- The data type:
  - Integer (OPL keyword **`int`**)
  - Real (OPL keyword **`float`**)
  - String (OPL keyword **`string`**)
- The data structure, which can be a scalar, a range, a set, an array, or a tuple.

An example of a simple OPL data declaration is:

```
float unitProfit = ...;
```

- **`float`** is the OPL keyword used for real (fractional) data or decision variables
- **`unitProfit`** is the name of the data item
- in this case, the data structure is a scalar – more complex data structures such as sets, arrays and tuples are indicated by a special syntax, which you'll learn about later.
- all data declarations end with `...;`, unless the data is initialized in the same line.

Data can be initialized in the model (a .mod file) or the data (a .dat file) files, or read from spreadsheets and databases. Data can also be exported from OPL to an Excel spreadsheet or a supported database. Spreadsheet or database read and write statements are written in .dat files.

A data element may be instantiated directly as input data or it can be computed in the model file. Computed data elements are sometimes referred to as variables but they are different from **decision variables** (in OPL models) or **script variables** (in IBM ILOG Script).

The term "data element," is used to distinguish data clearly from decision variables and script variables.

Decision variables

When declaring decision variables, you need to decide:

- The name of the variable
- The variable type:
  - Integer (OPL keyword **`int`**)
  - Real (OPL keyword **`float`**, for MP only)
  - Boolean (OPL keyword **`boolean`**)
  - Interval (OPL keyword **`interval`**, for CP only)
  - Sequence (OPL keyword **`sequence`**, for CP only)
- The data structure, which can be a scalar, a range, a set, an array, or a tuple.
- Optionally, the domain, which is the set of possible values the variable can take.

The variable type **`interval`** is new starting with OPL 6.0 and is used to model scheduling problems that are solved with IBM® ILOG® CP Optimizer. This variable type represents an interval of time during which an activity occurs, and is characterized by a start, an end, a size and an intensity.

An example of a simple OPL decision variable declaration is:

```
dvar float+ production in 0..maxCapacity;
```

- **`dvar`** is the OPL keyword used to declare decision variables
- the (optional) **+** sign is OPL syntax that indicates this variable can take only non-negative values
- **`production`** is the name of the decision variable
- **`in 0..maxCapacity`** defines the domain of the variable to include only values between 0 and the maximum capacity (**`maxCapacity`** being another data item)

Note that the + sign allows shortcut notation and that **`dvar float+ production;`** is equivalent to **`dvar float production in 0..infinity;`**. This can also be used with **`int`**, for example **`dvar int+ x;`** is equivalent to **`dvar int x in 0..maxint;`** where **`maxint`** is an OPL keyword representing the largest possible positive integer value.

Also note that **`dvar boolean x;`** declares a binary decision variable and is equivalent to **`dvar int x in 0..1;`**

Decision variable expressions

OPL decision variable expressions can be used to write more complex expressions in a compact way.

An example of a simple OPL decision expression is:

```
dexpr float+ profit =  production*unitProfit;
```

- **`dexpr`** is the OPL keyword used to declare decision expressions
- **`profit`** is the name of this particular decision expression
- this expression defines **`profit`** to equal the **`production`** decision variable multiplied by the **`unitProfit`** associated with each unit produced.

Using decision expressions modifies the number of variables, constraints, and nonzeros at execution time and can impact both the solution time and the memory consumption.

Objective function

When defining your objective function, you need to decide whether it's a maximization or minimization problem, as well as the expression you'd like to optimize.

An example of a simple OPL objective declaration is:

```
maximize profit;
```

- **`maximize`** is the OPL keyword used for maximization problems. **`minimize`** is used for minimization problems.
- **`profit`** is the decision expression to be maximized, in this case. The objective expression can be very complex or very simple, depending on your problem.

Alternatively, one could omit the **`profit`** decision expression and write this objective as follows:

```
maximize production*unitProfit;
```

If you intend to use your OPL model as part of an ODM application, it is recommended that the objective consists of a decision expression or the sum of decision expressions, because decision expressions in the OPL objective correspond to goals in ODM.

Having an objective function is optional – you can choose to find a solution that satisfies the constraints without optimizing a particular objective.

Constraints

Constraints in OPL are written in a block starting with **`subject to {`** and ending with **`}`**.

An example of a simple OPL constraint block containing a single constraint is:

```
subject to {
productionConstraint: production <= capacity;
}
```

- **`productionConstraint`** is the name, or constraint label, of this constraint. Labeling constraints is optional.
- this constraint states that the **`production`** quantity must be less than or equal to the **`capacity`**

OPL considers only labeled constraints for relaxation when attempting to resolve infeasibilities.

Instead of writing constraints within **subject to {...}**, you can use **constraints {...}**. These are equivalent.

OPL includes many keywords that you can use to define a wide variety of constraints for both MP and CP models.

Expressing variable bounds as a constraint

If you want to specify upper and lower bounds on a decision variable (or an expression containing a decision variable) *where the bounds themselves involve decision variables* then these upper and lower bounds must be expressed separately as two OPL declarations. For example, the following code is *not* allowed:

```
dvar int x in 0..5;
dvar int y;
dvar int z;
minimize x;
subject to {
 z <= y <= x;
}
```

Instead, you must write:

```
dvar int x in 0..5;
dvar int y;
dvar int z;
minimize x;
subject to {
 z <= y
 y <= x
}
```

Variables **x, y** and **z** are decision variables whose value is not yet calculated, therefore, upper and lower bounds must be specified separately.

However **z<=y<=x** is allowed in the case where **z** and **x** are not variables, but data elements. Such expressions using non-variable bounds can have at most three operands. For example, **w<=z<=y<=x** is not allowed.

Instructor note
This limitation on expressing bounds in constraints was introduced in OPL 5.0. The illegal example above would have been legal in OPL 4.x and earlier. Students who have used older versions of OPL may need to have this pointed out. This may be of particular interest to CP Scheduling users who are upgrading from OPL 3.7.

OPL data files

Learning objective
Learn what the contents of an OPL .dat file are, and how to write a simple data initialization statement.

Key terms
- .dat file
- data initialization
- data read
- data write

What's in a data file?

Data (.dat) files facilitate separation of the model and the data, thus allowing the use of several different data instances with the same model. Data files may include:

- Data initialization
- Statements to connect to spreadsheets and/or databases
- Statements to initialize data by reading values from spreadsheets and/or databases
- Statements to write solution values to spreadsheets and/or databases

In this topic you'll learn how to write a simple data initialization. More complex initialization (for example for arrays or sets) are covered together with the discussion on these data structures, while communication with spreadsheets and databases are covered in another lesson.

Some of the characteristics of data files are:

- Data initialization directly in a `.dat` file cannot contain computations or expressions of any kind, only raw symbolic or numeric data.
- Data types are not required in the data file, because these have already been defined in the model file. You simply use the name of the data item and assign a value to it.
- It is possible to use IBM ILOG Script in a `.dat` file to define custom ways of reading and formatting data.
- The data file and model file within the same project need not have the same name. You can use several different data files with a model file within the same project. You can also concatenate multiple data files in one project, or run different data sets on the same model in the same project by using run configurations.

Concatenated data files are processed in the order in which they are declared in a project.

Basic data initialization

Consider the following example of a simple OPL data declaration:

```
float unitProfit = ...;
```

This data can either be initialized in the model file by replacing the declaration with the following line:

```
float unitProfit = 2.5;
```

or you can keep the declaration as before in the model file and initialize the data item in data file as follows:

```
unitProfit = 2.5;
```

Notice that the data type is not used when initializing data in the data file. It's generally better to initialize data in the data file, as it keeps the model generic with respect to data.

OPL data structures

Learning objective
Learn what the basic OPL data structures are, and how to declare them. Learn how to use indices.

Key terms
- range
- set
- array
- tuple
- index

OPL data structures

In this topic you'll learn about the basic use of the OPL data structures:

- range
- set
- array
- tuple

Later in this lesson you'll learn how to combine these data structures for more versatility.

Ranges

Integer ranges are fundamental in OPL, because they are often used in arrays and decision variable declarations, as well as in aggregate operators, queries, and quantifiers.

An integer range is specified using the keyword **range** and giving its lower and upper bounds, as in

```
range Rows = 1..10;
```

which declares a range named **Rows** with bounds 1 and 10. The lower and upper bounds can also be given by expressions, as in

```
int n = 8;
range Rows = n+1..2*n+1;
```

An integer range is typically used:

- as an array index in an array declaration

  ```
  range R = 1..100;
  int A[R]; // A is an array of 100 integers
  ```

- as an iteration range

  ```
  range R = 1..100;
  forall(i in R) {
     //element of a loop
  }
  ```

- as the domain of an integer decision variable

  ```
  dvar int i in R;
  ```

You'll learn more about arrays and the **forall** statement later in this lesson.

You can also specify a range of type **float**. The declaration:

```
range float myFloatRange = 1.2..2.2;
```

specifies the subset of the real numbers in the interval [1.2,2.2].

Sets

OPL sets are non-indexed collections of elements without duplicates. OPL supports sets of arbitrary types to model data in applications. If **T** is a type, then **{T}**, or alternatively **setof(T)**, denotes the type “set of T”. For example, the declarations

```
{int} myIntegerSet = ...;
setOf(int) myIntegerSet = ...;
```

both declare a set called **myIntegerSet** containing elements of type integer.

Sets may be ordered, sorted, or reversed. By default, sets are ordered, which means that:

- Their elements are considered in the order in which they have been created.
- Functions and operations applied to ordered sets preserve the order.

A set is often used as an array index in an array declaration. For example, in the following code snippet we first declare a set called **Products**, and then use that set as an index for the decision variable array **production**.

```
{string} Products = {"product1","product2","product3"};
dvar float production[Products]; // production is a decision variable
 array indexed over the set of Products
```

A word on using indices

Before elaborating on how to initialize sets, it's important to understand the use of indices. You've already seen how a range or a set can be used as an array index. Generally, indices are used to concisely write similar expressions that only differ in the item(s) the expression is declared for. Instead of writing the expression for each item separately, one can simply use an index and define the expression over that index.

For example, without using an index, one would have to define production variables for three products as follows:

```
dvar float production_product1;
dvar float production_product2;
dvar float production product3;
```

If using the set **Products** as an index, this becomes:

```
dvar float production[Products];
```

You can also define a parameter to refer to each element of the set or range used as index, for example:

```
p in Products
```

Here, **p** is a parameter and **Products** is the set of data from which **p** takes its values. You can filter the index using a filtering condition:

```
p in Products : filtering condition
```

For example, you can declare an integer set of **ProductNumbers** to denote three products:

```
{int} ProductNumbers = {1, 2, 3};
```

Then, use that set to index two arrays of type **float**, namely **capacity** and **production**.

```
float capacity[ProductNumbers] = ...;
dvar float production[ProductNumbers];
```

Next, declare a constraint that uses the parameter **p** to refer to each element of the set **ProductNumbers**, so that you write the constraint only once, even though it's defined for all products (note the use of the OPL keyword **forall**):

```
subject to {
forall(p in ProductNumbers)
 production[p] <= capacity[p];
}
```

Finally, you can filter the products for which this constraint will be applied by using the filtering condition **`p <= 2`**:

```
forall(p in ProductNumbers : p <= 2)
 production[p] <= capacity[p];
```

Several indices can be combined in a comma-separated list to produce more compact statements. For instance,

```
int s = sum(i,j in 1..n: i < j) i*j;
```

is equivalent to

```
int s = sum(i in 1..n) sum(j in 1..n: i < j) i*j;
```

which is less readable.

OPL provides quantifiers and aggregators as tools that work with indexes to simplify model declarations.

Quantifiers:

- The **`forall`** quantifier is used in both MP and CP models. It is used to generate one constraint for each instance of the indexed entity. The declaration,

  ```
  forall (p in ProductNumbers)
    production[p] <= capacity[p];
  ```

  creates constraints that enforce the production of each product to be less than or equal to the production capacity for that product.
- The **`all`** quantifier is used in CP models to filter a set of objects to be used as arguments for certain CP constructs. This is discussed in more detail in another lesson.

Aggregators:

- Integer and float expressions can be constructed using aggregate operators for computing summations (**`sum`**), products (**`prod`**), minima (**`min`**), and maxima (**`max`**) of a collection of related expressions. For example, the objective function

  ```
     maximize sum (p in Products) profit[p] * production[p];
  ```

  illustrates the use of the aggregate operator **`sum`** to calculate the summation of the profit over all products.
- Here is an example of quantifier and aggregator in use together:

  ```
  forall (p in products)
    sales[p] == sum (l in locations) locSales[l][p];
  ```

Now that you've had an overview of how to use indices in OPL, let's return to the discussion of sets.

Initializing sets

Sets can be initialized in the following ways:

- Internally in the model file:

```
{int} myIntegerSet = {1, 3, 5, 7};
```

- Externally in the data file:

```
myIntegerSet = {1, 3, 5, 7};
```

- In a generic way using another range or set, for example use **`IntegerSet`** to create **`anotherIntegerSet`** with elements 1 and 3:

```
{int} anotherIntegerSet = {i | i in myIntegerSet : i <= 3};
```

In the latter statement, the notation "**`|`**" can be read as "such that", as in "create each element, **`i`**, of **`anotherIntegerSet`** *such that* **`i`** exists in **`myIntegerSet`** and on the condition that **`i`** is less than or equal to 3". The generic method of initializing sets is very powerful with expressiveness similar to relational database queries.

Explicit values assigned to set elements are always given inside curly brackets.

Some other examples of set initialization are:

- Using the **`asSet`** keyword to convert a range to a set:

```
{int} mySet = asSet(1..10);
```

- Using the modulus (**`mod`**) operator to create a set of every third number between 1 and 10:

```
{int} mySet = {i | i in 1..10 : i mod 3 == 1};
```

  This is equivalent to **`{int} mySet = {1, 4, 7, 10};`**.
- Using the **`union`** operator on two other sets to create a new set:

```
{int} mySet3 = mySet1 union mySet2;
```

Several other operators are available for set manipulation, for example **`subset`**, **`inter`**, and **`diff`**. You can read more about these in the OPL documentation.

Memory usage considerations

Sets instantiated by ranges are represented explicitly (unlike ranges). As a consequence, a declaration of the form **`{int} s = asSet(1..100000);`** creates a set where all the values 1, 2, ..., 100000 are explicitly represented.

This uses more memory than a simple range declaration. For example, the declaration

```
range s = 1..100000;
```

stores only the bounds (1 and 100000) explicitly.

In the discussions on ranges and sets, you saw some simple examples of arrays and perhaps you already have some idea of the OPL syntax used to declare arrays. In the next section, you'll learn more about OPL arrays.

Arrays

OPL arrays can be multidimensional. The array elements can be of the basic data types or more complex data structures:

- **`int`** , **`float`** or **`string`**
- Sets
- Tuples

This topic focusses on arrays of the basic data types, and you'll learn more about arrays of sets or arrays of tuples in a later topic.

Initializing arrays

An array structure is indicated by the use of square brackets, `[]`, around the array index. Arrays can be initialized in the following ways:

- In the model file:

  ```
  int myIntegerArray[1..4] = [1, 3, 5, 7];
  ```

- In the data file:

  ```
  myIntegerArray = [1, 3, 5, 7];
  ```

- In a generic way (known as generic arrays), for example:

  ```
  int anotherIntegerArray[i in 1..10] = [i+1];
  ```

  This declares an array of 10 elements such that the value of `a[i]` is `i+1`.

- In an IBM ILOG Script execute block (recommended for complex cases only), for example:

  ```
  range R = 1..8;
  int a[R];
  execute {
    for(var i in R) {
      a[i] = i + 1;
    }}
  ```

Note that the explicit values assigned to array elements are always given inside square brackets. When initializing the array in the .dat file, the array type and the dimensions are omitted.

Array indices

An array index can be:

- a range, for example

  ```
  float unitProfit[1..4] = ...;
  ```

- a set, for example

  ```
  float unitProfit[Products] = ...;
  ```

- defined by a generic expression (these are known as generic indexed arrays), for example

  ```
  int myArray[1..10] = [ n-1 : n | n in 90..99 ];
  ```

The latter statement creates an array of 10 elements. The expression before the colon (`n-1`) represents the value of the index, while the expression after the colon (`n | n in 90..99`) represents the value of the array element at that index. For example, when `n` equals 90, the index will equal 89, and so forth.

The difference between a generic array and a generic indexed array is that the former expresses only the array elements in a generic way, while the latter expresses both the indices and the array elements in a generic way.

Multidimensional arrays

The following is an example of a two-dimensional array declaration and initialization:

```
int my2DArray[1..2][1..3] = [ [5, 2], [4, 4], [3, 6] ];
```

This statement declares an array in two-dimensions, namely a 2 x 3 array, and initializes the values of each array element. The declaration and initialization can also be done separately with the declaration in the model file:

```
int my2DArray[1..2][1..3] = ...;
```

and the initialization in the data file:

```
my2DArray = [ [5, 2], [4, 4], [3, 6] ];
```

You can combine different types of indices in multidimensional arrays, for example:

```
int numberOfWorkers[Days][1..3] = ...;
```

declares a two-dimensional array whose elements are of the form **`a[Monday][1]`**. This can be used, for example, to indicate the number of workers on each day of the week for each of three daily shifts.

Some examples of generic multidimensional arrays are:

```
int m[i in 1..10][j in 0..10] = 10*i +j;
```

This initializes element **`m[i][j]`** to **`10*i + j`** and so on.

```
int m[dim1][dim2]=...;
int t[i in dim2][j in dim1] = m[j][i];
```

This transposes **`m`**.

Arrays of decision expressions

You can use arrays together with the keyword **`dexpr`** to create more compact, efficient models. For example, the declaration

```
dexpr int surplus[i in Stock] = stock[i] — demand[i]
```

creates an array that is handled very efficiently: because the definition is kept as a reusable object, it is not necessary to store every value of the expression for every index value.

Tuples

OPL provides advanced data types called tuples. A tuple is analogous to a row in a database table. For example, if your problem data consists of a demand for each product in each time period, you may have a database table with three columns, e.g. product, timePeriod, and demand. This data can be declared in tuple form as follows:

```
tuple productData{
 string product;
 int timePeriod;
 float demand;
}
```

In addition to this tuple declaration, you would typically also declare a set that uses this tuple as its type. An analogy to explain this is to think of a tuple referring to one row in the data table and you therefore need a set of tuples to access the entire table:

```
{productData} ProductData = ...;
```

Tuple elements can be of several types, including sets, arrays, and other tuples. This topic focuses on tuples with basic types as elements, while you'll see in the next topic how to combine tuples with some of the other data structures.

Tuple keys

As in database systems, tuple structures can contain one or more keys. Tuple keys enable you to access data using a set of unique identifiers, for example:

```
tuple nurse {
 key string name;
 int seniority;
 int qualification;
 int payRate;
}
```

In this code, the **`nurse`** tuple is declared with the key **`name`** of type **`string.`**

Using keys has the following advantages:

- The key field (in this case **`name`**) is a unique identifier. In the example above, in a set of tuples of type **`nurse,`** no two tuples can have the same **`name.`** If a user inadvertently attempts to add two different tuples with the same **`name,`** OPL will raise an error.
- Defining keys enables you to access elements of the tuple set by using only the value of the key field.

The following code shows a tuple with multiple keys:

```
tuple shift {
 key string departmentName;
 key string day;
 key int startTime;
 key int endTime;
 int minRequirement;
 int maxRequirement;
}
```

A shift is uniquely identified by the department name, the date, and start and end times, all defined as key fields. Both the above examples are taken from the **`nurse.prj`** file supplied in the **`<OPLhome>\examples\opl`** directory distributed with the product.

Initializing tuples

Tuples are initialized by listing the values of the various fields, within the delimiters "<" and ">". For example, if the following tuple has been declared:

```
tuple Point {
     int x;
     int y;
  };
```

it can be initialized:

- In the model file:

  ```
  Point p = <2,3>;
  ```

- In the data file:

  ```
  p = <2,3>;
  ```

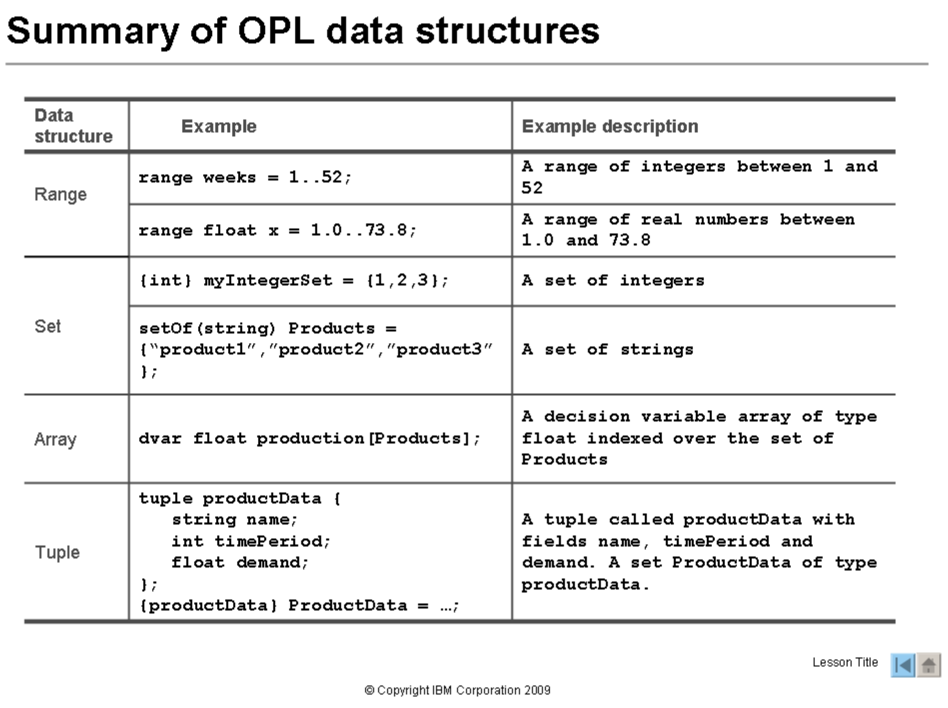
Tuple fields can be accessed by suffixing the tuple name with a dot and the field name, for example:

```
int x = p.x;
```

Note that the field names are local to the scope of the tuple.

OPL data structure summary

The following chart summarizes the OPL data structures, showing examples of the basic syntax. In a later topic you'll learn how to combine these data structures for more versatility.



Summary of OPL data structures

Data structure	Example	Example description
Range	`range weeks = 1..52;`	A range of integers between 1 and 52
	`range float x = 1.0..73.8;`	A range of real numbers between 1.0 and 73.8
Set	`{int} myIntegerSet = {1,2,3};`	A set of integers
	`setOf(string) Products = {"product1","product2","product3"};`	A set of strings
Array	`dvar float production[Products];`	A decision variable array of type float indexed over the set of Products
Tuple	`tuple productData { string name; int timePeriod; float demand; }; {productData} ProductData = …;`	A tuple called productData with fields name, timePeriod and demand. A set ProductData of type productData.

Lesson Title

© Copyright IBM Corporation 2009

A telephone production problem

Learning objective
Write a simple optimization model using OPL.

Key term
OPL model

A telephone production problem

If you attended the “Learning MP for OPL” training course, you may recall this problem. It is repeated here for those who did not attend the earlier course.

The business problem:

A telephone company processes and sells two kinds of products:

- Desk phones
- Cellular phones

Each type of phone is assembled and painted by the company, which wants to produce at least 100 units of each product and to maximize its quarterly profit.

Maximizing the quarterly profit becomes the objective function in the model.

In order to do this, the company has to calculate the optimal number of each type of phone to produce.

The number of each type of phone to produce are the decision variables in the model.

The conditions the company must work with are as follows:

- A desk phone's processing time is:
  - 12 min. on the assembly machine and
  - 30 min. on the painting machine.
- A cellular phone's processing time is:
  - 24 min. on the assembly machine and
  - 24 min. on the painting machine.
- The assembly machine is available for only 400 hours per quarter.
- The painting machine is available for only 490 hours per quarter.

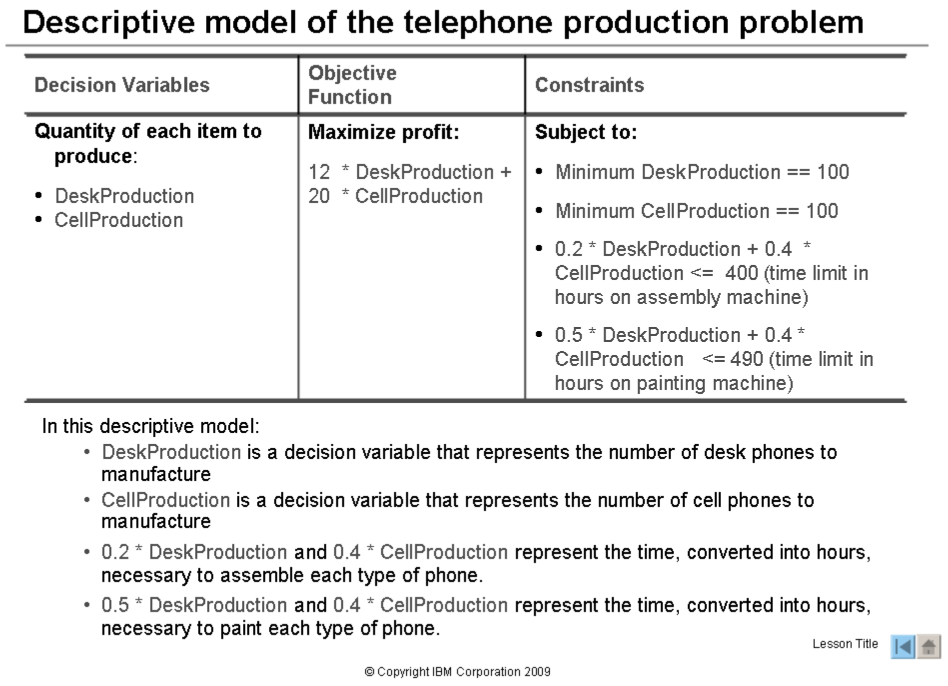
These conditions will be used to formulate constraints.

We also know the profit returned from sales of each type of phone:

- Desk phones return a profit of $12 per unit.
- Cellular phones return a profit of $20 per unit.

These data elements will be used to formulate the objective function.

The following is a descriptive model of this problem:



Descriptive model of the telephone production problem

Decision Variables	Objective Function	Constraints
Quantity of each item to produce: • DeskProduction • CellProduction	**Maximize profit:** 12 * DeskProduction + 20 * CellProduction	**Subject to:** • Minimum DeskProduction == 100 • Minimum CellProduction == 100 • 0.2 * DeskProduction + 0.4 * CellProduction <= 400 (time limit in hours on assembly machine) • 0.5 * DeskProduction + 0.4 * CellProduction <= 490 (time limit in hours on painting machine)

In this descriptive model:

- DeskProduction is a decision variable that represents the number of desk phones to manufacture
- CellProduction is a decision variable that represents the number of cell phones to manufacture
- 0.2 * DeskProduction and 0.4 * CellProduction represent the time, converted into hours, necessary to assemble each type of phone.
- 0.5 * DeskProduction and 0.4 * CellProduction represent the time, converted into hours, necessary to paint each type of phone.

Lesson Title

Instructor note
This is an opportunity to stress to students the importance of using a descriptive model. In programming terms, this phase is the equivalent for developers of writing pseudo-code.

Practice

Write the telephone production model in OPL

You are now going to create an OPL project to model and run the telephone production problem. The process is as follows:

1. Create a new project and write the model
2. Create a new data file in the project and separate the data from the model.
3. Add the model and data files to a run configuration.

Complete the **Telephone production** workshop. You can perform this lab using the HTML workshop, or by following the instructions in the workbook. The HTML workshop will give you direct access to OPL documentation pages that can help you with the lab.

Telephone production

Problem description

This lab takes you through a basic linear programming problem that demonstrates basics for any OPL model. It also shows how you separate data from the model in the OPL IDE.

A telephone company processes and sells two kinds of products:

- *Desk phones*
- *Cellular phones*

Each type of phone is assembled and painted by the company, which wants to produce at least 100 units of each product.

- *A desk phone's processing time is:*
  - *12 min. on the assembly machine and*
  - *30 min. on the painting machine.*
- *A cellular phone's processing time is:*
  - *24 min. on the assembly machine and*
  - *24 min. on the painting machine.*
- *The assembly machine is available for only 400 hours.*
- *The painting machine is available for only 490 hours.*
- *Desk phones return a profit of $12 per unit.*
- *Cellular phones return a profit of $20 per unit.*

The objective is to maximize profit.

Exercise folder

`<trainingDir>\OPL63.Labs\Phones\work`

This directory is empty when you start this lab. You are going to create a project in it.

Write model

Objective

- *Model with OPL*

References

arrays
floats

quantifier 'forall'
aggregate operator 'sum'

Actions

1. *Create a new project. Name it* **`phoneswork`** *and save it to* **`<training_dir>\OPL63.Labs\Phones\work`**.
2. *In the model file,* **`phoneswork.mod`**, *declare a set of strings for the product name data elements.*

   ```
   {string} Products = {"desk", "cell"};
   ```

3. *Declare arrays to specify the other static data elements.*

   All time units must be the same; minutes should be converted into hours.

   ```
   float Atime[Products] = [0.2, 0.4]; //assembly time

   float Ptime[Products] = [0.5, 0.4]; //painting time

   float Aavail = 400; //available time on assembly machine

   float Pavail = 490; //available time on painting machine

   float profit[Products] = [12, 20]; //profit realized
   from each product

   float minProd[Products] = [100, 100]; //minimum
   production for each product
   ```

4. *Declare an array of production decision variables. Use* **`dvar`** *to designate a decision variable*

   ```
   dvar float+ production[Products];
   ```

5. *Declare the expression to maximize*

   use **`sum`** *and* **`maximize`**

6. *Declare the constraints*

   use **`forall`** *and* **`subject to {}`**

7. *Save the project.*

Separate data from model

Objective

- *Manipulate data initialization*

Actions

- *Action 1: create and fill .dat*
- *Action 2: update .mod*

References

executing a project
data initialization
labeling constraints

Action 1: create and fill .dat file

- *Save* **`phoneswork.mod`** *under the name* **`phones1.mod`**.

Right click on the ***phonesWork*** *project and select* ***refresh*** *from the context menu, or type* **`F5`**

- *Create a new run configuration. By default, it will be named* **`Configuration2`**.
- *Add file* **`phones1.mod`** *to* **`Configuration2`** *by dragging the file to the configuration in the OPL Projects Navigator.*
- *Create new data file* **`phones1.dat`** *in the project.*
- *Add file* **`phones1.dat`** *to* **`Configuration2`**.
- *Instantiate all the data elements in* **`phones1.dat`**.

Do not include type information from the model file

OPL provides the "..." escape sequence to separate a data declaration from its instantiation. For example, **`float Pavail = 490;`** *becomes* **`float Pavail = ...;`** *. The data type is not given in the data file, for example the data file will contain* **`Pavail = 490;`***, as opposed to* **`float Pavail = 490;`**

It might become difficult to read, control, or maintain a data file if a large number of values are listed. To allow better readability, OPL provides a named instantiation syntax. It is available for arrays or tuples indexed with sets:

- *The "#" symbol is used to start and end named array instantiations.*
- *The symbol ":" separates the name from the value.*

Example: **`profit = #[desk:12 cell:20]#;`**

Action 2: update .mod file

- *In the* **`phones1.mod`** *file, change the instantiation of data elements so that instance values are taken from the* **`phones1.dat`** *file. Use the "`...`" construct.*
- *Run Configuration2.*

Note how run configurations can be used to allow different versions of model and data to coexist in a single project.

Solutions

Actions

- *Write model*
- *Action 1: create and fill .dat*
- *Action 2: update .mod*

Write model

Solutions file:

`<trainingDir>\OPL63.Labs\Phones\solution\phonesSolution`

```
/*********************************************
 * OPL Model file
 * Author: IBM ILOG
 * Creation date: 3/7/2006 1:48 PM
 *********************************************/
```

```
//declare a set of products
{string} Products = { "desk", "cell" };

//declare data
float Atime[Products] = [0.2, 0.4];
float Ptime[Products] = [0.5, 0.4];
float Aavail = 400;
float Pavail = 490;
float profit[Products] = [12, 20];
float minProd[Products] = [100, 100];

//declare an array of decision variables
dvar float+ production[Products];

//declare objective function and constraints
maximize
   sum (p in Products) profit[p] * production[p];
subject to {
   forall (p in Products)
      production[p] >= minProd[p];
      sum (p in Products) Atime[p] * production[p] <= Aavail;

      sum (p in Products) Ptime[p] * production[p] <= Pavail;


}
```

Actions 1: create and fill .dat file

```
Products = { "desk" "cell" };
Atime = [0.2 0.4];
Ptime = [0.5 0.4];
Aavail = 400;
Pavail = 490;
minProd = [100 100];
profit = [12 20]
```

Actions 2: update .mod file

```
/*********************************************
 * OPL Model file
 * Author: IBM ILOG
 * Creation date: 3/7/2006 1:48 PM
 *********************************************/

// declare data
{string} Products = ...;
float Atime[Products] = ...;
float Ptime[Products] = ...;
float profit[Products] = ...;
float Min[Products] = ...;
float Aavail = ...;
float Pavail = ...;

// declare decision variables
```

```
dvar float+ production[Products];

// declare objective function and constraints
maximize
   sum (p in Products) profit[p] * production[p];
subject to {
   forall (p in Products)
      production[p] >= minProd[p];

      sum (p in Products) Atime[p] * production[p] <= Aavail;

      sum (p in Products) Ptime[p] * production[p] <= Pavail;


}
```

Modification to **`.dat`** *file for named instantiation:*

```
profit = #[desk:12 cell:20]#;
```

You have now built and solved your first OPL model! In the next topic you'll learn how to combine OPL data structures for increased versatility.

Combining OPL data structures

Learning objective
Learn how to combine different data structures. Understand especially how to use sets of tuples.

Key term
data structures

In this topic, you'll learn how to combine the different data structures available in OPL for more versatile use. Specifically, you'll learn how to use:

- Sets inside tuples
- Arrays inside tuples
- Tuples inside tuples
- Sets of tuples
- Arrays of tuples
- Arrays of sets

You'll also learn about some useful data structure manipulations, namely:

- Converting an array to a tuple set
- Using sets to instantiate arrays

Sets inside tuples

Sets can be used as tuple elements, for example:

```
tuple Members {
  int memberNumber;
  {string} memberInterests;
};
```

This declares a tuple type **Members** containing a tuple element for the **memberNumber**, as well as a tuple element for the set of the member's interests.

Arrays inside tuples

One-dimensional arrays can be used as tuple elements, for example:

```
tuple ProductData {
     int productId;
     float insideCost;
     float outsideCost;
     float consumption[Resources];
}
```

This code declares a tuple type **ProductData** consisting of 4 elements:

- **productId** of type **int**
- **insideCost** and **outsideCost** of type **float**
- **consumption[Resources]** - an array of type **float**

You can declare and initialize a tuple of this type as follows:

```
ProductData p = <1, 0.52, 0.17, [0.7, 0.9]>;
```

While arrays of **int** and **float** are permitted as tuple elements, you cannot use them if you intend to instantiate your tuple by reading from a spreadsheet or database.

Thus, it is considered a practice to avoid when the model will require data from a spreadsheet or database, for example when the model is used as part of an ODM application.

Tuples inside tuples

A tuple can have one or more tuples as elements. For example, consider a case where you'd like to associate products with production plants. Then you may have an additional tuple type for the production plant data:

```
tuple PlantData {
    int plantId;
    float capacity;
 }
```

and associate the products with the production plants in a tuple type called **`ProductAtPlant`**:

```
tuple ProductAtPlant {
    ProductData product;
    PlantData plant;
 }
```

If a tuple key element has another tuple as a member, and no elements of the subtuple are declared as keys, OPL will assume that all members of the subtuple are keys. If some members of the subtuple are explicitly declared as keys, only those members will be treated as keys.

Using tuples is a powerful way to ensure that only the essential data is instantiated. In the preceding example, the tuple type **`ProductAtPlant`** allows the model to only consider particular product-plant combinations, as opposed to associating all products with all plants, which may not be valid. Using tuples to create sparse data instances is critical for large problems where memory use may become a major issue affecting solution time.

Some limitations apply to the contents of tuples:

- Multidimensional arrays are not allowed.
- Arrays of strings, arrays of tuples and arrays of tuple sets are not allowed.
- Sets of tuples (instances of IloTupleSet) are not allowed.

Sets of tuples

Once a tuple type T has been declared, you can declare sets of tuples of type T. Consider again the **`ProductData`** tuple type defined earlier:

```
tuple ProductData {
     int productId;
     float insideCost;
     float outsideCost;
     float consumption[Resources];
}
```

Using this tuple type, you can declare and initialize a set of tuples as follows:

```
{ProductData} pDataSet = {<1, 0.52, 0.17, [0.7, 0.9]>, <2, 0.21,
0.44, [0.6, 0.3]>};
```

The set **`pDataSet`** contains two tuples of type **`ProductData`**. When initializing tuples as in the examples above, each tuple instance included in the set is written inside angle brackets (< >).

Arrays of tuples

You can declare and initialize an array of tuples of type **`ProductData`** as follows:

```
ProductData pDataArray[i in 1..5] = <i, 2*i, 2.5*i, [0.7, 0.9]>;
```

This example declares an array of 5 tuples of type **`ProductData`**, where the values of **`insideCost`** and **`outsideCost`** change, depending on the value of the **`productId`** (which in this particular case corresponds to the array index).

Arrays of sets

OPL supports arrays of sets, for example:

```
{int} myIntArray[1..2] = [{1,2},{3,4}];
```

It is also possible to initialize an array of sets in a generic way. For example, the declaration:

```
{int} a[i in 3..4] = {e | e in 1..10: e mod i == 0};
```

instantiates **`a[3]`** to {3,6,9} and **`a[4]`** to {4,8}.

Converting an array to a tuple set

It is possible to convert data represented by an array to a tuple set by using a generic set initialization. For example, in the following code, the 2D Boolean array, **`edges`**, describes the edges of a graph:

```
{string} Nodes ...;
int edges[Nodes][Nodes] = ...;
```

This array can be transformed into a set of edges, where each edge is represented by a tuple, as follows:

```
tuple Edge {
 Nodes o;
 Nodes d;
}
{Edge} setEdges = {<o,d> | o,d in Nodes : edges[o][d]==1};
```

Using tuple sets is highly recommended, because tuple sets can be used to easily create a **sparse representation** of the data. In this example, the number of elements of the array **`edges`** equals the square of the number of nodes (that is, all possible edges for all possible nodes), while the tuple set **`setEdges`** contains only those elements for which an edge actually exists (**`edges[o][d] == 1`**). Sparsity is discussed in more detail later in this lesson.

Using sets to instantiate arrays

Sometimes it is useful to use a set in order to instantiate an array in a generic way. The following code extract shows how this can be done:

```
{string} Gasolines = ...;
{string} Oils = ...;
{gasType} GasData = ...;
{oilType} OilData = ...;
gasType Gas[Gasolines] = [ g.name : g | g in GasData ];
oilType Oil[Oils] = [ o.name : o | o in OilData ];
```

This code comes from one of the models included in the OPL distribution (**`<OPLhome>\examples\opl\oil\oilDB.mod`**). The sets **`GasData`** and **`OilData`** are temporary sets that are initialized externally by reading data from a database. These sets are used later in the model to declare the one-dimensional **`gas`** and **`oil`** arrays.

In the first array declaration, the text to the left of the colon, **`g.name`**, indicates the index to use for the array (indexed by the set of **`Gasolines`**), namely the **`name`** attribute of each tuple in the set of **`GasData`**. The text to the right of the colon, **`g | g in GasData`**, indicates the corresponding array element, namely the relevant tuple. In other words, **`Gas[g.name] = g.`**

By analogy, **`Oil[o.name] = o`** in the second declaration.

Using sets to instantiate arrays is a powerful tool that can be used for creating arrays from large tuple sets.

General OPL syntax

Learning objective
Understand the general syntax for declaring OPL model elements.

Key term
syntax

Throughout this lesson, we used simple examples to explain OPL syntax. In this topic you'll see general syntax for declaring some of the OPL elements, specifically:

Instructor note
While the general syntax may not be of interest to all students, you may choose to discuss this in more detail if students are interested. There are no slides on this topic, as it doesn't completely cover OPL syntax and may be confusing to some.

- Tuples
- Decision variables
- Objectives

While you do not need to use this type of general syntax, it may be of interest to you if you have, for example, a computer science background.

Tuples

```
tuple <Tuple_type> {
<type> <Component_name>[<array_indexer(s)>];
...
}
```

Here, [ ] represent the grammar meta syntax meaning "optional" (a tuple component can optionally be an array).

Decision variables

```
dvar <type> <name> [=<shared dvar>];
| dvar <type> <name> [<indexer>][[<indexer>]...];
| dvar <type> <name> [<index in indexer>][[<index in indexer>]...]
 [=<shared dvar>];
```

Here, **dvar** and **in** are OPL keywords. The square brackets around **<indexer>** and **<index in indexer>** represent array syntax, while when there are two sets of square brackets, the inner set indicates array syntax, while the outer set indicates optional syntax. The square brackets around **=<shared dvar>** mean optional. The "|" symbol means "or" – in other words, the syntax for **dvar** shows three different ways of declaring a **dvar**.

Objectives

```
{ maximize | minimize } <expression>;
```

Here, { and } indicate grammar meta syntax, i.e. an objective can start with either **maximize** or **minimize**.

Sparsity and slicing

Learning objective
Learn about sparse data structures using tuple sets, implicit and explicit slicing, and performance monitoring.

Key terms
- sparsity
- slicing
- tuple set

Sparsity is defined as the fraction of zeros in a matrix, i.e. a sparse matrix has a large number of zeros compared to non-zeros.

Large linear programs tend to be very sparse, with sparsity increasing as the dimensions get larger.

Slicing is used to exploit sparsity in a model.

In slicing, you create pairs of values that identify cells of interest. Empty or irrelevant cells are ignored. You then use these pairs of values (and only these) to set up processing loops, data and/or constraints.

An example of slicing, taken from computer programming, is *array slicing,* in which certain elements from an array are extracted and repackaged as a different array, possibly with a different number of indices (or dimensions) and different index ranges. Two common examples are extracting a substring from a string of characters (e.g. "par" from "sparsity"), and extracting a row (or a column) of a rectangular matrix to be used as a vector.

Sparse data structures

A typical optimization model contains data structures that are in effect combinations of other data structures, such a tuple that includes other tuples as elements, or a multi-dimensional array that uses a different set to index each dimension. For example, you may want to declare a set of Products and a set of Plants and then declare another data structure to indicate which products can be manufactured at which plants.

Not all possible combinations are valid, however. For example, not all products can be manufactured at all plants. Removing these invalid combinations from the matrix eliminates the need to create, store and iterate unnecessary values.

The result is:

- Time is saved — only valid data is processed by the engine
- Memory is saved — unnecessary values are not stored and take up no memory space.

Memory is especially critical in large models: if a model requires more physical memory (RAM) than is available, the computer swaps some of its memory to the hard disk (“virtual memory”), and the constant movement back and forth as different parts of memory are swapped in and out dramatically increases running time. For maximum performance, the model needs to be fully loaded into physical memory with enough space left over for calculation.

Even when virtual memory is not necessary, compact data structures allow larger models to be solved with the same running time as less efficient ones.

The best way to create a sparse model is to design it from the ground up using data structures and techniques that reduce redundant data. OPL provides several mechanisms that help you make efficient models with sparse data structures. The most important are:

- Tuple sets
- Slicing

The essential tool for monitoring memory use and running time is the **Profiler** output tab in the IDE.

Use tuple sets to instantiate only valid combinations

The following code declares a tuple type **workTask** that contains pairs of strings connecting workers to tasks:

```
tuple workTask {
 string workerID;
 string taskID;
};
```

A model for a large project might be dealing with thousands of workers and thousands of tasks. It is clearly irrelevant to process combinations of every worker with every task. A worker trained as a carpenter, for example, is most likely not qualified to do the job of an electrician, so such a pairing has no meaning for our model. Therefore, we want to create a subset of all the worker-task pairings, which includes only the valid ones. We do this with the following declaration in the model file:

```
{workTask} WorkerTasks = ...;
```

With this declaration, we create a set named **WorkerTasks** composed of members of the tuple **workTask**. The right hand side of the equation indicates that the data for this set is to be instantiated in the data file. These values can be specified in the data file itself, but they can also be called from an external source such as a spreadsheet or database. Connecting to these will be discussed in a later lesson. Here are some sample set member values:

```
WorkerTasks = {
<"GWashington" "Plumbing">
<"GWashington" "Carpentry">
<"NBonaparte" "Carpentry">
<"NBonaparte" "Painting">
<"WChurchill" "Plumbing">
<"WChurchill" "Painting">
};
```

Again, the **WorkerTasks** set only contains an element if it is possible to assign a worker to a task. There is no **<"GWashington", "Painting">** pair because the worker **GWashington** is not qualified to do the painting task.

Create a sparse array by indexing on the tuple set

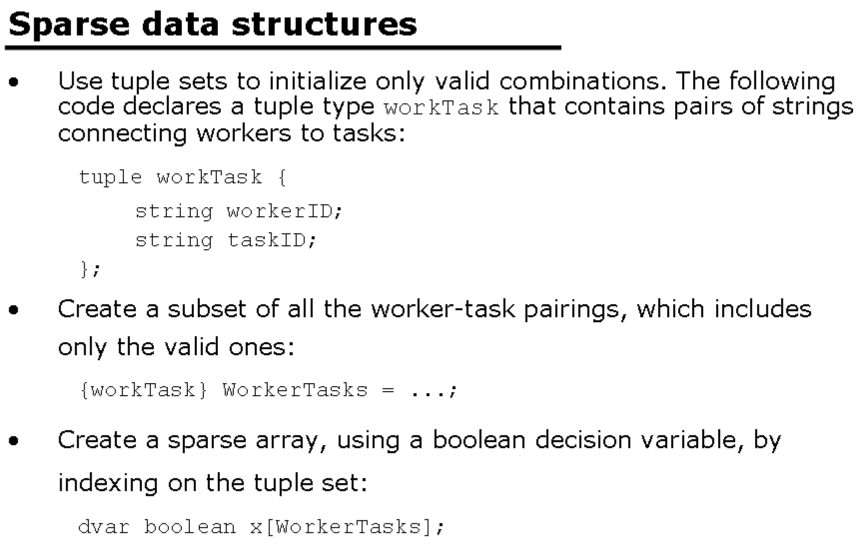
Consider a boolean decision variable **x**. This variable is limited to only those assignments of workers to task that are meaningful, as it is indexed on the sparse set **WorkerTasks:**

```
dvar boolean x[WorkerTasks];
```

Compare this to:

```
dvar boolean x[workerID] [taskID];
```

This code instantiates the array **x** with *all* possible workers performing *all* possible tasks. This is much less efficient.



Sparse data structures

- Use tuple sets to initialize only valid combinations. The following code declares a tuple type `workTask` that contains pairs of strings connecting workers to tasks:

```
tuple workTask {
     string workerID;
     string taskID;
};
```

- Create a subset of all the worker-task pairings, which includes only the valid ones:

```
{workTask} WorkerTasks = ...;
```

- Create a sparse array, using a boolean decision variable, by indexing on the tuple set:

```
dvar boolean x[WorkerTasks];
```

The efficient model uses slicing

Consider the following constraints:

```
forall (t in Tasks)
 assignTasks: sum (<w,t> in WorkerTasks) x[<w,t>] == 1
forall (w in Workers)
 assignWorkers: sum (<w,t> in WorkerTasks) x[<w,t>] == 1
```

The first constraint (**assignTasks**) ensures that every task is covered: for each task, it sums, indexed over the workers who are capable of performing the task. The second constraint (**assignWorkers**) ensures that workers are assigned to exactly one task: for each worker, it sums, indexed over the tasks that the worker is capable of performing.

These constraints, like the decision variable declaration, are written using **slicing**, which streamlines iteration over sparse sets.

Instead of iterating over all the pairs of workers and tasks, this code sample only iterates over the valid pairs specified in the sparse set **WorkerTasks**. Through slicing, the **assignTasks** constraint finds the matching **<w,t>** pair for each **t** in **Tasks**, and the **assignWorkers** constraint finds the matching **<w,t>** pair for each **w** in **Workers**.

To sum up, the decision variable declaration saves memory by only creating decision variables (**x**) for valid pairs of workers and tasks, while the constraint declarations save time and memory by only iterating over valid pairs of workers and tasks.

Explicit and implicit slicing

Consider a transportation problem where products must be shipped from one set of cities to another set of cities. The model may include a constraint specifying that the total shipments for all products transported along a connection may not exceed a specified limit:

```
forall(c in connections)
 sum(<p,co> in routes: c == co) trans[<p,c>] <= limit;
```

This constraint states that the total products **`p`** shipped along each connection **`c`** is not greater than **`limit.`** OPL must scan the entire set **`routes`** to select the tuples that meet the filtering condition, **`c==co`**. In this example, the filtering expression **`c==co`** is used to make slicing **explicit.**

The same constraint can be expressed with **implicit slicing** as follows:

```
forall(c in connections)
 sum(<p,c> in routes) trans[<p,c>] <= limit;
```

In this constraint, the tuple **`<p,c>`** uses the previously defined parameter **`c.`** Since the value of **`c`** is known, OPL uses it to index the set **`routes,`** avoiding a complete scan of the set **`routes.`**

In this example, slicing is said to be implicit because the index **`c`** is used to declare iteration in both the **`forall`** and **`sum`** loops. You can also use a constant as a tuple item, for example **`<p,2>,`** for implicit slicing.

In general, there is no performance advantage in using implicit slicing over explicit slicing. Therefore, you should use whichever renders your code more readable for you and your collaborators.

Instructor note
In previous editions of this training course, there has been a lab that uses the networking model from the Pasta Production workshop to demonstrate use of sparse data in this spot. Due to reorganization of the material in this edition, the networking lesson will not normally fit before this material. That means that the lab for sparsity will not come until later on, a bit distant from the theoretical material. This will be corrected in a future edition of this training.

A pasta production model

Learning objective
Practice using more advanced data structures, such as tuples, to create an optimization model for pasta production.

Key term
tuple

The business problem

To meet the demands of its customers, a pasta company chooses either to manufacture its products in its own factories (inside production) or to contract them from other companies (outside production). It does this as a function of demand and cost.

The inside production is subject to some resource constraints: each product consumes a certain amount of each resource. In contrast, outside production is only limited by contractual agreement with the suppliers.

The problem is to determine how much of each product should be produced inside and outside the company while minimizing the overall production cost, meeting the demand, and satisfying the resource and contractual constraints.

Requirements:

- Manufacture products from available ingredients
- Meet the customer demand
- Produce yourself (inside) or obtain from a subsidiary (outside) at different costs
- Satisfy resource availability constraints (stock on hand) for the inside production
- Do not exceed the contractual limits on outside production
- Minimize the overall cost

Details of the problem

The **Products**, different types of pasta, are:

- kluski
- capellini
- fettucine

The **Resources** used to produce the Products are:

- flour (stock on hand = 120 units)
- eggs (stock on hand = 150 units)

The **Consumption** of Resources for each product (in units) is:

- For a package of kluski
    - 0.5 flour
    - 0.2 eggs
- For a package of capellini
    - 0.4 flour
    - 0.4 eggs
- For a package of fettucine
    - 0.3 flour
    - 0.6 eggs

Customer **Demand** for each product is:

- 100 packages of kluski
- 200 packages of capellini
- 300 packages of fettucine

The **Inside cost** to make each product is:

- $ 0.60 per package of kluski
- $ 0.80 per package of capellini
- $ 0.30 per package of fettucine

The **Outside cost** to make each product is:

- $ 0.80 per package of kluski
- $ 0.90 per package of capellini
- $ 0.40 per package of fettucine

Finally, the **maximum Outside production** is contractually limited to be no more than 200 units per product.

Practice

Define a descriptive model of this problem that will enable you to write the OPL model.

Tasks:

1. Define the data that you will declare
2. Choose the decision variables to use
3. Define the objective function
4. Define the constraints

Later in this lesson, you will reorganize the data for this problem using the tuple data structure, but for now, just define the data elements you need to declare in the model.

Instructor note
Work with the students as they develop this problem. Guide them to use decision variable names that resemble or are identical to those used in the **`productionSolution`**project files, since they will be executing that workshop at the end of the lesson.

Practice

Model the pasta production problem in OPL

Go to the **Pasta Production and Delivery** workshop and complete the first step, **Write a basic model.** You can perform this lab using the HTML workshop, or by following the instructions in the workbook. The HTML workshop will give you direct access to OPL documentation pages that can help you with the lab.

Write a basic model

Objectives

- *Manipulate and initialize data*
- *Write constraints.*

Action

- *Action 1: finish production.mod*

Finish the production.mod file

Import the project **`productionWork`** *into the OPL Projects Navigator and look at the .mod file.*

- *The* **`consumption`** *array is indexed on* **`Products`** *and* **`Resources`**.
- *The* **`availability`** *array is indexed on* **`Resources`**.
- **`demand`** *and* **`cost`** *arrays (inside and outside) are indexed on* **`Products`**.
- *The decision arrays (insideProduction and outsideProduction* **`dvar`** *expressions) are indexed on* **`Products`**.

You can easily see the correspondence between the arrays as defined in the model, and the contents of the data file:

```
Products =  { "kluski" "capellini" "fettucine" };
Resources = { "flour" "eggs" };

consumption = [ [0.5, 0.2], [0.4, 0.4], [0.3, 0.6] ];
availability = [ 120, 150 ];
demand = [ 100, 200, 300 ];
insideCost = [ 0.6, 0.8, 0.3 ];
outsideCost  = [ 0.8, 0.9, 0.4 ];
maxOutsideProduction = 200;
```

- *Complete the following steps to write the model:*
    1. *Write the objective:*
       The sum, for each product, of the **`insideCost`** *times the* **`insideProduction`** *, plus the* **`outsideCost`** *times the* **`outsideProduction`** *.*

    2. *Write the constraint for all resources:*
       The sum of the **`consumption`** *times the* **`insideProduction`** *is less than or equal to the* **`availability`** *of the* **`Resource`**.

    3. *Write the constraint for all products:*
       The **`insideProduction`** *plus the* **`outsideProduction`** *is greater than or equal to the* **`demand`**.

- *Save and run your model.*
- *Compare your work to the solution found in* **`<trainingDir>\OPL63.labs\Pasta\Tuples\solution\productionSolution.`**

Practice

Model the pasta production problem using tuples

Go to the **Pasta Production and Delivery** workshop and complete the second step, **Write a model using tuples**. You can perform this lab using the HTML workshop, or by following the instructions in the workbook. The HTML workshop will give you direct access to OPL documentation pages that can help you with the lab.

Write a model using tuples

Objective

- *Practice using the tuple data structure for the pasta production model*

Actions

- *Create a new model using tuple structures*
- *Create a tuple without an internal array*

References

tuples
right-click context menu commands

From array to tuple

If you look at the original arrays declared in the previous step, the only one that is not related to the products is **`availability`**. *The availability limitations, indexed on the resources, will remain as an array.*

Convert the model to one that uses tuples

1. *Create a new project,* **`productwork`** *and save it to the work directory. The* **`productwork.mod`** *file is automatically created. Ask for the automatic creation of a data file,* **`product.dat`** *in the same project.*
2. *In the model file, use a tuple to hold all the information about each product. You will need to declare demand, inside and outside cost and resource consumption inside the tuple.*
3. *Redefine the objective function and constraints to take advantage of the tuple.*

   You need to provide access to individual members of the tuple in your objective function.

4. *Change the initialization in the data file: use the original data, but order it to initialize the new tuples.*

   *Use named initialization (***`#[`***) in the* **`.dat`** *file for clarity.*

5. *Run the program.*
6. *Compare your work to the solution found in* ***`<trainingDir>\OPL63.labs\Pasta\Tuples\solution\productSolution.`***

Create a tuple without an internal array

In the solution you just completed, the tuple element **`consumption`** *is an array. If you later need to import data into this tuple element from a spreadsheet or database, this structure cannot be used (OPL's spreadsheet and database connections are explained in another lesson with a corresponding workshop). In this step, you discover a way to avoid this problem.*

Arrays inside tuples are permitted, but you can use them only if you're not getting your data from a spreadsheet or database.

Steps to change the model:

1. *Make a copy of the* ***`<trainingDir>\OPL63.labs\Pasta\Tuples\work\productwork`*** *project by selecting the project name and typing* ***`<Ctrl>C`*** *followed by* ***`<Ctrl>V`*** *in the OPL Projects Navigator. When the* ***Copy Project*** *popup window appears, change the project name to* ***`product2work`****.*
2. *Open the file* ***`product2work.mod`*** *for editing.*
3. *The array inside the tuple concerns the consumption of resources. The first step, then, is to remove the array* ***`consumption`*** *from the tuple* ***`ProductData`****.*
4. *Define a new tuple, named* ***`consumptionData`*** *specifying, for each kind of pasta, how much of each resource is necessary.*

   To use this new structure and maintain the same solution, you need the next three steps.

5. *Create a set of this tuple, and name it* ***`consumption`****.*
6. *In the data file, remove the initialization of the* ***`consumption`*** *tuple element (which you removed in the model file) and reorganize the same data to initialize the new tuple set, also called* ***`consumption`****, that you just created.*
7. *Adapt the* ***`resourceAvailability`*** *constraint so that it uses the new data structures to express the same constraint as in the previous version of the model.*
8. *Run your model.*
9. *Compare your work to the solution found in* ***`<trainingDir>\OPL63.labs\Pasta\Tuples\solution\product2Work.`***

Summary

Review

In this lesson, you learned how to write a model in OPL using OPL syntax, together with the available OPL data structures and operators.

An OPL model typically contains:

- The choice of solver engine
- Data declarations
- Decision variables
- Objective function
- Constraints

An OPL model can also contain IBM ILOG Script statements for pre- and postprocessing, as well as flow control.

The basic data types available in OPL are:

- integer (OPL keyword **`int`**)
- real (OPL keyword **`float`**)
- string (OPL keyword **`string`**)

You learned what the data structures available in OPL are, as well as how to declare, initialize, combine and manipulate them:

- range
- set
- array
- tuple

In addition, you learned what the types of variables available in OPL are:

- integer (OPL keyword **`int`**)
- real (OPL keyword **`float`**, for MP only)
- Boolean (OPL keyword **`boolean`**)
- interval (OPL keyword **`interval`**, for CP only)
- sequence (OPL keyword **`sequence`**, for CP only)

You learned the meaning of sparsity and how to use tuples to create sparse model instances, and completed exercises that involved writing two models in OPL.

Lesson 3: Working with IBM ILOG Script: basic tasks

Instructor note
This lesson should last about 1 hour, including the practice. In the future this lesson will include a demo to show how to use IBM® ILOG® Script for flow control (a topic covered in detail in another lesson). While including a demo is planned for the next training release, you can choose to show such a demo at the end of this lesson if you have one available.

IBM ILOG Script is embedded in OPL to provide scripting functionality, including:

- Preprocessing data in the `.mod` file
- Postprocessing data in the `.mod` file
- Processing and formatting data in the `.dat` file
- Setting algorithmic parameters for both CPLEX® and CP Optimizer, including specifying a search phase for CP Optimizer.
- Flow control, for example to implement decomposition schemes (where a model is decomposed into a set of smaller more manageable models)

IBM ILOG Script can be added to any OPL model or data file.

At the end of this lesson you will be able to:

- Initialize an array in an IBM ILOG Script **`execute`** block
- Perform pre- or postprocessing using IBM® ILOG® Script **`execute`** blocks
- Prepare data using IBM ILOG Script **`prepare`** blocks

Using IBM ILOG Script to specify a search phase for CP Optimizer is covered in the lesson on **Solving Simple CP Problems**, while flow control is covered in detail in the (optional) lesson on **Flow Control with IBM ILOG Script**.

About IBM ILOG Script

Learning objective
Gain a general understanding of the role of IBM® ILOG® Script relative to the OPL modeling language

Key term
IBM ILOG Script

What is IBM ILOG Script?

A script is a sequence of commands. These commands can be of various types: declarations, simple instructions, and compound instructions.

IBM ILOG Script is an embedded JavaScript™ implementation to handle the *non-modeling* needs of OPL. It supports a variety of methods to:

- Solve the model
- Access its data
- Change the value of its settings
- Manage preprocessing and postprocessing
- Manage flow control
- Display data

IBM ILOG Script uses the same high-level data structures as in OPL, embeds them in an imperative language, and adds some novel constructs that address these non-modeling processes.

Both variable and data declarations are supported in IBM ILOG Script. You will find a complete reference for the IBM ILOG Script keywords in the OPL online help, in **Language Quick Reference > IBM ILOG Script Keywords.**

IBM ILOG Script basics

Learning objective

Learn the basics of how an IBM® ILOG® Scriptt is organized and the general syntax used

Key terms

- execute block
- main block
- prepare block

IBM ILOG Script provides three types of instruction blocks:

- **`execute`** blocks are used in the **`.mod`** file for preprocessing and postprocessing, and to set the CP search phase
- A **`main`** block is used in the **`.mod`** file for flow control (only one **`main`** block is allowed per model file)
- **`prepare`** blocks are used in the **`.dat`** file to perform additional operations related to data initialization

IBM ILOG Script is embedded in the OPL modeling language. All declared model elements are available for scripting via their name.

Script variables

Script variables are defined in **`main`** and **`execute`** blocks using the **`var`** keyword, and are local to the blocks in which they are declared.

The use of the word” variable” here is identical to the usual meaning of the word “variable” in a computer programming context. Note that this is different from a decision variable (in OPL models) or a data element that represents data in an OPL model.

Preprocessing and postprocessing

Learning objective

Understand how IBM® ILOG® Script works together with OPL language to process data before and after the model is solved.

Key terms

- preprocessing
- postprocessing

Preprocessing and postprocessing

A block of statements for preprocessing or postprocessing is marked by the keyword **`execute.`** The general form of an **`execute`** block is as follows:

```
execute <block_name> {
 <statement>;
 <statement>;
 ...
 <statement>;
}
```

where **`<statement>`** is any legal IBM ILOG Script instruction or declaration. An **`execute`** block can have one or more **`<statement>`**s.

The **`<block_name>`** is optional. It simply puts a handle on the block so it is more easily callable.

Any **`execute`** block placed *before* the objective or constraints declaration is part of preprocessing; other blocks are part of postprocessing.

No two execute blocks can have the same **`<block_name>`** within the same model.

Preprocessing

In OPL, the term **preprocessing** can refer to one of the following:

- Instructions that prepare your data for modeling and solving.
- The preprocessing done by the CPLEX® engine — i.e. a two-stage process composed of:
  - *Presolving* to reduce the size of a problem by making inferences about the nature of any optimal solution to the problem
  - *Aggregating* to eliminate variables and rows through substitution

IBM ILOG Script is used for the first operation mentioned above, deploying instructions that manipulate or prepare your data before the optimization model is created. Here is an example where all workers' salaries are adjusted before running the model:

```
execute {
   for (var w in Workers) {
      w.salary = w.salary * w.raise;
   }
}
```

The script variable **`w,`** declared inside the **`execute`** block using the **`var`** command, is local to the **`execute`** block and is therefore unknown to the OPL model and to other **`execute`** blocks.

For decision variables and data, the context within an **`execute`** block corresponds to the model declarations, in other words the set of **`Workers`** referred to here is the same as the one declared in the model.

Changing algorithmic parameters

IBM ILOG Script **`execute`** blocks can be used to set CPLEX or CP Optimizer parameters, such as parameters that control the presolve, solution display, choice of algorithm, etc.

An example of setting CPLEX parameters:
```
execute CPX_PARAM {
  cplex.preind = 0;
  cplex.simdisplay = 2;
}
```

This turns CPLEX presolve off and displays iteration information for each iteration of the simplex optimizer.

You will find a complete list of CPLEX parameters in the OPL documentation at **Language > Parameters and settings in OPL > Mathematical programming options > IBM ILOG CPLEX Parameters Reference Manual**.

An example of setting CP Optimizer parameters:
```
execute CP_PARAM {
 cp.param.TimeLimit = 13;
 cp.param.LogVerbosity = "Quiet";
}
```

In the example above, IBM® ILOG® CP Optimizer will deliver the best solution it has found after searching for 13 seconds, and will not output any information to the search log.

You will find a complete list of CP Optimizer parameters that you can call in IBM ILOG Script in the OPL documentation at **Language > Language User's Manual > IBM ILOG Script for OPL > Using IBM ILOG Script in constraint programming**.

You can also specify a search phase for CP Optimizer using an IBM ILOG Script **`execute`** block. This is covered on the lesson on **Solving Simple CP Problems**.

Parameters for IBM ILOG CPLEX and CP Optimizer can also be set in the settings (**`.ops`**) file.

For a complete list of parameters available in the settings file, refer to the documentation at **Language > Parameters and settings in OPL**.

Postprocessing

Postprocessing is used to manipulate solutions and control output. One of the most common postprocessing tasks is data display.

The following code fragment uses the **`writeln`** keyword inside an **`execute`** block to display data. **`execute`** blocks can be named. In the example that follows, the execute block is named **`STORES.`** Note that this script example uses the data objects **`Storesof`** and **`Stores`** which have been declared in the model (outside the script).

```
{int} Storesof[w in Warehouses] = { s | s in Stores : Supply[s][w]
 == 1 };
execute STORES{
    writeln("Storesof=",Storesof);
}
```

Another common postprocessing function is to obtain the objective value of the current solution, using the **`getObjValue()`** method. This value may then be processed further in the **`execute`** block or be output to the **Scripting Log**.

Data display

Data display is done with the **write** and **writeln** keywords. When used to display data or results in pre or postprocessing, the **writeln** keyword must be within an **execute** block:

```
execute {
   writeln("hire list = ");
   for (var h in hires)
      writeln(h);
}
```

The **writeln** command is also useful for inserting a blank line, which must be written as follows:

```
writeln()
```

The command **writeln** without parenthesis will not work.

writeln will put all elements of a set or array on the same line unless the data structure is being looped through in a **for** loop, as in the example above.

One could also use the problem solution to create and populate new data items that can be displayed using an **execute** block. In the example below the new data set **crew** is defined as the subset of an existing set of **workers** that should be hired according to the model solution, where **hire[c]** is a Boolean decision variable indicating whether a worker should be hired (1) or not (0).

```
{int} crew = {c | c in workers : hire[c] == 1}
 execute {
  for (var c in crew)
   writeln(c);
 }
```

Data initialization

Learning objective

Understand how IBM® ILOG® Script can be used to initialize arrays

Key term

initialization

Array initialization

As previously noted, the **`execute`** block can be used as an alternative method for initializing arrays. In the example below script is used to initialize a **`string`** array called **`names`** and an **`int`** array called **`salaries`**. The initialization assigns values to the array elements by looping over the set of **`Workers`**, where **`Workers`** is a set of **`tuples`** with **`name`** and **`salary`** fields, and assigning the **`name`** and **`salary`** values to the respective arrays.

```
string names[Workers];
int salaries[Workers];
execute {
 for(var w in Workers){
  names[w] = w.name;
  salaries[w] = w.salary;
}
```

This method is less efficient than a generically initialized array, and is recommended primarily for complex cases or multiple initializations.

Guidelines for array initialization using IBM ILOG Script

- Loop through the set corresponding to the values you need to initialize your data. In *Example 1* below, data from the set of **`TableRoutes`** is used to initialize the **`cost`** array. In this case, **`TableRoutes`** is a set of tuples.
- Assign a value to each array element based on the attributes of the set being looped through. In the example below the **`cost`** attribute of each tuple in the set of **`TableRoutes`** is used to populate the **`cost`** array.
- When assigning values to an array indexed by a tuple, use the **`find`** method of the **`tuple`** set:

  Example 1:

  ```
  execute INITIALIZE {
    for( var t in TableRoutes) {

  cost[Routes.find(t.product,Connections.find(t.origin,t.destination))]
   =   t.cost;
      }
   }
  ```

Here is another example:

Example 2:

```
tuple tupleType {int a; int b; int c; };
   {tupleType} tupleSet = {<1,2,3>};
   tuple tupleIndexType {int a; int b;};

   {tupleIndexType} tupleIndex = {<a,b> | <a,b,c> in tupleSet};

   int arrayIdxByTup [tupleIndex];
   execute OPTIONAL_NAME {
      for ( var x in tupleSet )
          arrayIdxByTup[tupleIndex.find(x.a,x.b)] = x.c
   }
```

It is worth emphasizing, again, that this method should be reserved for situations where the initialization is very complicated. For example, it is also possible to initialize the array in *Example 2* using a generically indexed array as follows:

```
int arrayIdxByTup2 [tupleIndex] = [<a,b>:c | <a,b,c> in tupleSet];
```

This method is preferable to using the **`execute`** block.

Processing values in the .dat file

Learning objective
Understand how to manipulate data in the `.dat` file before the model is solved.

Key term
prepare block

You can use certain IBM® ILOG® Script operations to process values in the **`.dat`** file at initialization time. This is useful, for example, when your source data is in one form and you need to have it in another to use in the model. To do this, you use the **`prepare`** keyword to create a block in which you define the operation to be performed. The operation defined in the **`prepare`** block is called in the initialization statement using the **`invoke`** keyword.

The following code extracts show a simple example of how you do this. The model file contains a declaration for a data element **`t`** that represents time, expressed in hours:

```
float t[1..3]=...;
```

However, all the time data that is to be used to initialize this declaration is given in minutes. Either you have to calculate the conversion yourself, or, you can write the following IBM ILOG Script in the data file:

```
prepare {
  function transformIntoHours(t) {
      for(var a=0;a<t.length;a++) t[a]=t[a]/60;
      return true;
   }
}
```

Then, when you initialize **`t`**, you also call the conversion function:

```
t = [600,240,150] invoke transformIntoHours;
```

This declaration initializes **`t`** to values of 10, 4, and 2.5, i.e. the declared values converted from minutes to hours.

You can combine this type of operation with data importation from an external source such as a database.

Each **`.dat`** file can contain at most one **`prepare`** block.

Flow control

Learning objective

Understand what an IBM® ILOG® Script **`main`** block is used for.

Key term

main block

What is flow control?

Flow control enables you to control how models are instantiated and solved, for example to:

- solve several models with different data in sequence or iteratively
- run multiple "solves" on the same base model, modifying data and/or constraints after each solve
- decompose a model into smaller more manageable models, and solve these to arrive at a solution to the original model (model decomposition)

Some examples where you may want to use IBM ILOG Script for flow control are:

- Implementing column generation (a mathematical programming algorithm)
- Decomposing a supply chain model into a planning model and a scheduling model and solving these in sequence

Flow control can also be implemented using the available OPL APIs, for example for implementations in C++ or Java™.

The main block

Flow control statements are written within an IBM® ILOG® Script **`main`** block. An example of a simple **`main`** block is:

```
main {
 model.generate();
 cplex.solve();
}
```

- **`model.generate()`** is used to generate the OPL model
- **`cplex.solve()`** calls the CPLEX solver engine to solve the model
- **`main`** blocks can be written either before the objective or after the constraint block

Flow control is covered in detail in the optional lesson on **Flow Control with IBM ILOG Script**.

Each **`.mod`** file can contain at most one **`main`** block.

Summary

Review

In this lesson, you learned the basic functions of IBM ILOG Script, an embedded JavaScript implementation to handle the scripting needs of OPL. In this lesson, you learnt how to use IBM ILOG Script in OPL to:

- Preprocess data in the **`.mod`** file
- Postprocess data in the **`.mod`** file
- Process and format data in the **`.dat`** file
- Set algorithmic parameters

You also learned that an IBM ILOG Script **`main`** block is used for flow control, for example to implement decomposition schemes.

Next steps

Using IBM ILOG Script to specify a search phase for CP Optimizer is covered in the lesson on **Solving Simple CP Problems**.

Flow control is covered in detail in the (optional) lesson on **Flow Control with IBM ILOG Script**.

Lesson 4: Solving Simple LP Problems

Instructor note
This lesson should last about 1 hour 30 minutes, including the practices.

In this lesson, you will learn more about the OPL functionality available for Linear Programming (LP) problems, and use OPL to model and solve an LP problem. You'll practice using OPL data structures, especially tuples and sets.

In the process, you will become more familiar with the different data structures available in OPL, and especially the powerful capability of the tuple data structure that organizes data in a clear and coherent manner for easy manipulation in your model.

The lesson takes you through the steps that will enable you to start with a problem expressed in business terms, and arrive at a well-formed mathematical representation of the problem.

This lesson includes:

- An overview of some of the OPL functionality available for LP problems
- Practice converting a problem given in business terms to an mathematical model
- A lab to give you practical experience writing and solving an LP model using OPL

At the end of this lesson you will be able to:

- Use some additional OPL functionality available for constructing LP models, beyond what was covered in the preceding lessons
- Write an LP model in OPL

LP modeling structures

Learning objective
By the end of this lesson, you will have learned more about specific modeling structures used in MP models in OPL, and have practiced writing an LP model.

Key terms
- operators
- ordered indices
- constraint labels

In this topic

In this topic, you'll learn about more of the OPL functionality available for constructing Linear Programming (LP) models, specifically:

- Numeric and symbolic operators
- Using ordered indices
- Logical constraints
- Labeling constraints

The functionality covered in this topic is also applicable to Mathematical Programming (MP) problems, in general.

Operators

OPL provides a rich set of operators for MP models in an intuitive format. This workbook provides a series of tables listing the operators available for linear programming.

Numeric operators:

Type of operator	Operator	Description
Float	`+,-,*,/`	The usual mathematical operators
	`infinity`	Represents the infinite (IEEE 754)
	`abs(f)`	The absolute value of **f**
	`ceil(f)`	The smallest integer greater than or equal to **f**
	`floor(f)`	The largest integer less than or equal to **f**
	`trunc(f)`	The integer part of **f**
	`frac(f)`	The fractional part of **f**
	`distToInt(f)`	The distance from **f** to nearest integer
	`round(f)`	The nearest integer to **f**
Integer	`+,-,*,div`	The usual mathematical operators
	`x mod y` or `x % y`	The integer remainder of **x** divided by **y**
	`abs(x)`	The absolute value of **x**
	`maxint`	The greatest integer

Symbolic operators:

Type of operator	Operator	Description
Set	**`first, last`**	First (last) element in the set list
	`item`	The n-th element in the set
	`card`	Number of elements in the set
	`ord`	Integer rank of an element in the set (zero based)
	`next, prev`	Next (previous) element in the set list
	`nextc, prevc`	Circular versions of **`next`** and **`prev`**
	`a inter b`	Intersection
	`a union b`	Union
	`a diff b`	Difference
	`a symdiff b`	Symmetrical difference (i.e. all elements that exist in (a+b) - (a union b))
String	**`==`**	Equivalent to
	`!=`	Different from
Boolean	**`&&`**	Conjunction (logical "and")
	`||`	Disjunction (logical "or")
	`==`	Equivalence
	`!=`	Difference (exclusive "or")
	`!`	Negation (logical "not")

Instructor note
Refer students to the online help for additional keyword explanations. Some of these operators are also available in CP Optimizer. They may be available only for filtering in CPLEX® but permitted on decision variables in CP Optimizer.

Ordered indices

To enforce indexing to take place according to the order of items in the set, use the notation:

```
forall (ordered i, j in positions)
```

This is equivalent to the statement:

```
forall (i,j in positions : ord(S,i)<ord(S,j))
```

Logical constraints in MP

In CPLEX models, a logical constraint combines linear constraints by means of logical operators, such as logical-and, logical-or, negation (not), conditional statements (if ... then ...) to express complex relations between linear constraints.

IBM® ILOG® CPLEX can also handle certain logical expressions appearing within a linear constraint. One such logical expression is the minimum of a set of decision variables. Another such logical expression is the absolute value of a variable.

In OPL, you can define logical constraints using any operator that can be extracted by the optimization engine. In addition to the relationships described above for linear constraints,

IBM ILOG CPLEX can extract the following logical operators in constraints:

- && (conjunction)
- | | (disjunction)
- ! (negation)
- != (difference)

All these constructs accept as their arguments other linear constraints or logical constraints, so you can combine linear constraints with logical constraints in complicated expressions in your application.

Constraint labels

The OPL IDE lets you attach a label to a constraint. This helps identify the constraint, and it is also good practice for the following reasons:

- Constraint labels enable you to benefit from the expand feature in the IDE Problem Browser to find which constraints are tight in a given application or to find dual decision variable values in linear programs.

  Refer to **Getting Started with the OPL IDE > Getting Started tutorial > Examining a solution to the model > Understanding the Problem Browser** in the documentation for details on how to do this.

- When a solution is available, you can access the slack and dual values for labeled constraints using IBM ILOG Script. IBM ILOG Script is discussed in another lesson.

  See also, the **IBM ILOG Script Reference Manual** in the documentation.

- Only labeled constraints are considered by the relaxation and conflict search process in infeasible models - these are discussed more fully later in this training.

Instructor note

Named constraints, used in versions of OPL before 5.0 can no longer be used. Use constraint labels instead.

To label a constraint, just type the name you want, followed by a colon (:), before the constraint you want to label.

```
minimize
   sum( p in Products)
     ( InsideCost[p] * Inside[p] + OutsideCost[p] * Outside[p] );
```

```
subject to {
  forall( r in Resources )
   ctCapacity:
    sum( p in Products )
     Consumption[p][r] * Inside[p] <= Capacity[r];

  forall(p in Products)
   ctDemand:
    Inside[p] + Outside[p] >= Demand[p];
}
```

In the example above, **`ctCapacity`** and **`ctDemand`** are the constraint labels.

Supermarket display problem

Learning objective
Model a real-world linear programming problem in OPL.

Key terms
- tuple
- array
- linear program

Practice

Managing supermarket display space
You are now going to perform a lab that takes you through a basic linear programming problem and shows how you can:

- Retrieve raw data from an external data source
- Pre-process the data before working on a decision model
- Post-process the data before returning a solution

In the process, you will practice using tuples, arrays and sets and practice instantiating them.

The pre-processing and post-processing in this lab are done using IBM® ILOG® Script **`execute`** blocks.

The business problem

A supermarket needs to optimize the display of available goods on storage shelves in order to maximize sales and profit. The supermarket sells different kind of products:

- Italian Food:
  - Pasta
  - Tomato Sauce
  - Salami
- Asian Food:
  - Rice
  - Soya Sauce
  - Chicken Wings

Some of these products have a sales interest apart from the ethnic food groupings above. Chicken wings, for example, are not only interesting to customers wanting to make Asian food. The store manager has to decide how much of each product to put in normal, open shelving and how much to put grouped with related products in special promotional groups as shown above.

In addition, there are constraints based on the floor plan of the supermarket and the different types of shelving available for these products.

Attributes of products and shelving

Each type of product is characterized by the following attributes:

- The expected unit profit margin
- The unit volume
- Must be refrigerated or not
- The minimum available quantity to be sold
- The maximum quantity that may be ordered (and hence sold)

The products can be displayed in different storage shelves across the supermarket. The shelves are limited in volume, and some are refrigerated, others not.

The different shelf units have different values for promoting the sales of items, as a function of the floor plan. These shelves have the following attributes:

- A volume capacity
- Is refrigerated? (Yes/No)
- A sales acceleration/efficiency factor (for example, higher for placement at the end of an aisle)

The sales acceleration factor has the effect of clustering certain items together on the “super promotion” shelves.

Go to the **Supermarket Display** workshop and perform the step, **Model the input data**. You can perform this lab using the HTML workshop, or by following the instructions in the workbook. The HTML workshop will give you direct access to OPL documentation pages that can help you with the lab.

Model the input data

Objective

- *Use different types of set instantiation to model the input data of this problem*

Actions

- *Define the products and their attributes*
- *Define product supply*
- *Define the shelving and its attributes*

References

sets
tuples
initializing sets
initializing tuples

Define the products and their attributes

When integrating the optimization engine into a legacy system, the input data may already be available in a determined format the engine must comply with, such as an ascii file, a csv file, an Excel spreadsheet, a data base, memory resident objects, etc.

In this case, for simplicity, we simply use plain ascii text data in the **`.dat`** *file to instantiate the model.*

Product attribute matrix

Product name	*Profit margin*	*Unit volume*	*Needs refrigeration?*
Tomato Sauce	*1.1*	*1*	*No*
Salami	*1.5*	*1*	*Yes*
Rice	*5*	*1*	*No*
Soya Sauce	*5*	*1*	*No*
Chicken Wing	*1*	*1*	*Yes*

Model this structure with a tuple:

```
tuple Product
{
key string name; // Product name.
 float margin; // Profit margin
 float unitVolume; // Volume for one unit of product
  int cold; // 1 if refrigeration required , 0 otherwise
};
```

Note that the "Needs refrigeration?" column (yes/no or true/false values) of the matrix is represented as a binary as follows:

- *1 if refrigeration is required*
- *0 if no refrigeration is required (for "dry" products)*

Collect all the members of this tuple together into a set:

```
{Product} products = ...;
```

Define product supply

Product supply matrix

Product name	***Minimum units available***	***Maximum possible order***
Tomato Sauce	*100*	*1000*
Salami	*50*	*300*
Rice	*100*	*1000*
Soya Sauce	*100*	*1000*
Chicken Wing	*40*	*1000*

Model this structure with the following tuple:

```
 tuple Supply
 {
  key string product;
  int minimumStockValue; // minimim quantity to be ordered
for stock/sale
  int maximumStockValue; // potential maximum extra ordered
quantity
 }
```

Collect all the members of this tuple together into a set:

```
{Supply} supplies = ...;
```

Define the shelving and its attributes

Shelving attribute matrix

Name	***Maximum Capacity***	***Sales Accelerator Factor***	***Refrigerated?***
SuperPromoFridge	*100*	*1.5*	*Yes*
StandardShelf	*1000*	*0.5*	*No*
PromoFridge	*100*	*1*	*Yes*
PromoShelf	*1000*	*1*	*No*
StandardFridge	*100*	*0.5*	*Yes*
SuperPromoShelf	*1000*	*1.5*	*No*

Model this structure with the following tuple:

```
 tuple Shelf
 {
  key string name; // shelf name
  int volumeCapacity; //maximum capacity
 float  promotionAccelerator; // indicator between 0 and 1.5

  int cold; // 1 for refrigerated, 0 otherwise
}
```

Collect all the members of this tuple together into a set:

```
{Shelf} shelves = ...;
```

What are the unknowns?

- How much of each product to display on each shelf?
- How much extra product (beyond stock on hand) should be ordered?

Modeling the decision variables

In this model, we could define a decision variable matrix in order to express all possible product quantities which are displayed into all storage shelves. This is, however, not a good idea; we know, for instance, that it is not possible to display a refrigerated product on a non refrigerated shelf.

It is better to create an array of decision variables **only** for compatible **`product-storageShelf`** pairs.

This means the array of decision variables must be indexed with such pairs. You will represent the pairs with a tuple.

In the **Supermarket Display** workshop, perform the step, **Model the decision variables**.

Model the decision variables

Objective

- *Use sparse sets to calculate values for decision variables during modeling*

Actions

- *Define compatibilities*
- *Declare the decision variables*

References

union
expressions of decision variables

Define compatibilities

You need to define a decision variable to determine what quantity of which product is displayed on which shelf. It will be indexed on pairs of product and shelf. Since products are compatible with only one type of shelf, some product/shelf pairs can be eliminated:

- *Cold products can only be displayed on cold (refrigerated) shelves*
- *Dry products can only be displayed on dry (non-refrigerated) shelves*

1. *To define a **sparse** set of compatible pairs, declare a tuple:*

```
tuple ProductShelfCompatibility
{
 Product product;
 Shelf shelf;
}
```

2. *Create two computed sets of instances of* **`ProductShelfCompatibility`***:*
   - *One containing only cold product/shelf pairs named* **`coldCompatibilities`**
   - *One containing only dry product/shelf pairs named* **`dryCompatibilities`**
3. *Declare a third set,* **`compatibilities`***, that is the aggregate of the two sets you declared in the last step.*

   Use the **`union`** *operator.*

Next you will use these pairs as indexes for displayed product and stored product decision variables. This is a simple way to create a sparse matrix under OPL

Declare the decision variables

1. *Use the set* **`compatibilities`** *as an index over decision variables for displayed product quantities over shelves. call it* **`storedQuantities`***:*

   ```
   dvar float storedQuantities[compatibilities] in
   0..maxint;
   ```

2. *Define a decision expression called* **`orderedQuantities`** *that uses the set* **`compatibilities`** *as an index over variables for product total ordered quantities*

   ```
   dexpr float orderedQuantities[ p in products] =
   //complete this expression using array initialization
   ```

What is the objective?

Maximize sales by displaying the products in the right places and the right quantities.

In the **Supermarket Display** workshop, perform the substep **Write the objective functions** of the step **Write the objective function and constraints**.

Write the objective function

In the **`.mod`** *file, write an objective function to maximize profit:*

1. *For each product/shelf compatibility pair, define the profit as a function of:*
   - *Displayed quantity for the shelf*
   - *Product's profit margin*
   - *The shelf's sales accelerator factor*

   Use a decision expression, using the keyword **`dexpr`***. Name it* **`profit`**

2. *Maximize* **`profit`***.*

What are the constraints?

- At least the minimum stock on hand must be displayed
- The total quantity ordered for each product must not exceed the maximum allowed order
- Total shelf capacity cannot be exceeded.

In the **Supermarket Display** workshop, perform the substep **Write the constraints** of the step **Write the objective function and constraints**.

Write the constraints

Two of the constraints are already completed for you.

1. *The first already completed constraint specifies that for all products, the sum of the ordered quantities shall be at least equal to the minimum stock value (stock on hand):*

```
forall( p in products, s in supplies: s.product ==
p.name){
 MinStockCst: orderedQuantities[p] >=
s.minimumStockValue;
}
```

2. *A constraint is completed requiring that for all products, the sum of the ordered quantities shall not exceed the maximum ordered quantity:*

```
forall( p in products, s in supplies: s.product ==
p.name){
 MaxStockCst: orderedQuantities[p] <=
s.maximumStockValue;
}
```

3. *Write a constraint requiring that for all shelves, the total quantity of all the displayed products multiplied by the product unit volume shall not exceed the total shelf volume capacity.*

Visualizing the results

In order to display the results in a useful manner, you need to create structures that can be manipulated after solving the model.

What do we want to see?

- The quantity of each product allocated to each shelf. To do this, declare a tuple, `StorageResult`, that associates shelf, product displayed on the shelf, and quantity of that product to be displayed on the shelf.
- How much extra of each product to order: declare a tuple, `PurchaseOrderResult`, associating the product and the amount to be ordered.
- The percentage of shelf space in use for each shelf: declare a tuple, `ShelfUsage`, associating each shelf with a usage percentage.

Use input data and decision variable values to calculate sets of these tuples that show results for all products and shelves.

This data could be exported to an external application such as a database or spreadsheet, for further processing and analysis. Here, for simplicity, we write the information to the **Scripting log** output tab using an IBM ILOG Script `execute` block. This feature is discussed in detail in another lesson of this training.

In the **Supermarket Display** workshop, perform the step, **Display the results**.

Display the results

Objective

- *Use sets to post-process data for display*

Actions

- *Create output data structures*
- *Convert structures into formatted results for display*
- *Experiment with the values*

Create output data structures

In the model, the output tuples are already declared for you:

- ***`StorageResult`***
- ***`PurchaseOrderResult`***
- ***`ShelfUsage`***

*The **`usagePercentage`** member of this tuple is to be calculated "on the fly" as part of the set of shelf usage ratios (see next substep).*

Examine these structures in the model.

Convert structures into formatted results for display

Complete the data structures for conversions into formatted results. Use generic set instantiation for this, following the descriptive patterns suggested below:

- *Ordered quantities for each product:*

```
 P = sum(  P/all shelf compatibilities)
orderedQuantities(P)
```

- *Quantity of displayed products on a given shelf:*

```
Q = sum( all product/S compatibilities)
storedQuantities(product/S)
```

- *Shelf usage ratio for a given shelf:*

```
S= sum( all product/S compatibilities)
storedQuantities(product/S) / S.capacity
```

Compare your work with the solution in
`<trainingDir>\OPL63.Labs\Mart\solution\martSolution`

Experiment with the values

In principle, if shelves are not full, the model will change the number of products to be ordered so that all shelf space is used 100%. However, there may be limits of budget or product availability that make it so that the maximum permitted order of some products would be exceeded.

*Try reducing the value of **`maximumStockValue`** for some or all products, and see what results you get when solving the model. Compare several different sets of values to see how OPL deals with the changed data.*

*Examine the displayed results in the **Scripting log** output tab, and the values of data elements and decision variables in the **Problem browser**.*

Summary

Review

In this lesson, you learned about some OPL modeling tools used in MP models:

- Operators available in OPL
- Logical constraints in MP models
- Constraint labels

You have also used these data structures to solve a production outsourcing problem and a supermarket product-shelf allocation problem.

Lesson 5: Solving Simple CP Problems

Instructor note
This lesson should take about 2 hours, including the practices.

Some optimization problems are solved very effectively using a technique called **Constraint Programming (CP)**. IBM® ILOG® CP Optimizer is a constraint programming optimizer for solving detailed scheduling problems as well as certain combinatorial optimization problems that cannot be easily linearized and solved using traditional mathematical programming methods. This powerful CP engine is easily accessible through OPL, where the user can take full advantage of OPLs development, debugging and tuning features to solve complex real-world CP problems.

A major benefit of IBM ILOG CP Optimizer is that it automatically generates advanced algorithms based on the mathematical formulation of a particular model. This allows users to “model and run” complex problems, without having to write the complex searches traditionally required for CP. However, it still allows the advanced user to specify a search if desired. In addition, it provides easy-to-use representations for specialized constraints. These simplified representations are particularly useful when tackling, for example, complex scheduling problems.

This lesson introduces you to basic CP concepts, and shows you how to take advantage of this technology using IBM ILOG CP Optimizer and OPL.

At the end of this lesson you will be able to:

- Understand what constraint programming is
- Model a simple CP problem using OPL
- Specify a CP search phase in OPL

IBM ILOG CP Optimizer is IBM's second-generation CP engine and is distinguishable from IBM's first-generation CP engine called IBM ILOG CP. While IBM ILOG CP is not embedded in OPL, and therefore not covered in this training, it remains available for complex routing problems that are beyond the current capabilities of IBM ILOG CP Optimizer.

Introduction to CP

Learning objective
Understand the principles of Constraint Programming and how it is implemented in OPL and CP Optimizer

Key terms
- Constraint Programming (CP)
- satisfiability
- combinatorial optimization
- constraint propagation
- domain reduction
- search strategy
- search phases

What is Constraint Programming?

Constraint programming is a discipline of computer science, closely associated with artificial intelligence. Formally, it is based on logic and symbolic reasoning. It is a technique that is very effective in solving, for example:

- Detailed scheduling problems
- Routing problems
- Satisfiability problems
- Certain other combinatorial optimization problems not well-suited for traditional Mathematical Programming (MP) methods

Historically, the word “programming” as used in “Constraint Programming” refers to a computer programming paradigm, as opposed to referring to a problem or methodology as it does in the case of “Mathematical Programming”. It is important not to confuse CP as being another branch of MP, even though these two techniques may seem to be tackled in a similar manner when using the OPL IDE. Specifically, in both cases OPL allows the user to build a model using OPL modeling syntax, to solve the model using a built-in IBM ILOG solver engine (CPLEX® for MP and CP Optimizer for CP), and to utilize other OPL functionality such as scripting.

CP Optimizer and OPL

IBM ILOG CP Optimizer is intended for the following two categories of problems, with the most important technical differences indicated below:

- Detailed scheduling problems
    - Decision variables are of type **`interval`** and describe an interval of time during which, for example, a task occurs
    - There is no discrete enumeration of time
- Certain combinatorial optimization problems not well-suited for MP
    - These problems contain only discrete decision variables

IBM ILOG CP Optimizer as embedded in OPL has the following additional benefits:

- Automatically generated advanced algorithms based on the mathematical formulation of a model
- Easy-to-use representations for specialized constraints used in scheduling and other combinatorial problems

Scheduling with CP Optimizer

Scheduling can be seen as the process of optimally assigning start and end times to intervals. Scheduling problems also require the management of minimal or maximal capacity constraints for resources over time.

IBM ILOG CP Optimizer as embedded in OPL provides elements to concisely represent complex scheduling problems, for example:

- Variables of type **`interval`**, with attributes of start, end, size and intensity
- Precedence constraints, for example **`endBeforeStart`** to indicate that the end of one interval must occur before the start of another
- Cumulative expressions to define resource constraints, for example **`cumulFunction, step`** and **`pulse.`**
- Other elements to model sequencing, synchronization, etc.

Note that a separate lesson on **Scheduling in OPL with CP Optimizer** provides a thorough overview on this subject. The current lesson focuses on using CP Optimizer to solve other types of combinatorial problems.

Combinatorial optimization with CP Optimizer

Certain combinatorial optimization problems cannot easily be linearized or solved using traditional mathematical programming techniques and are instead well-suited for CP. The elements of these problems are as follows:

- A set of discrete decision variables (integer or Boolean)
- A predefined domain for each variable designating its possible values
- A set of constraints defined according to the rules of CP
- Optionally, an objective function to be minimized or maximized

Models where no particular objective is being minimized or maximized, are known as **satisfiability** or **constraint satisfaction** problems. Here the user simply wants a *feasible* solution that satisfies all constraints over a set of decision variables. CP is especially useful for such problems.

You will explore the use of CP Optimizer for combinatorial optimization, other than scheduling, in more detail later in this lesson.

How does CP Optimizer work?

CP methodology, in general, has two phases:

1. Write a model representation of a problem in a computer programming language
2. Describe a search strategy for solving the problem

A typical CP search uses the following techniques iteratively until a solution is found:

- Domain reduction: The process of eliminating possible values from a variable domain by considering the constraints on that variable and related variables.
- Constraint propagation: The process of communicating the domain reduction of a decision variable to all of the constraints on this variable. This can result in more domain reductions.

CP Optimizer offers built-in search strategies based on your problem formulation, and you therefore don't need to write your own search strategy. However, it still allows you to specify a search if desired.

Constructive search

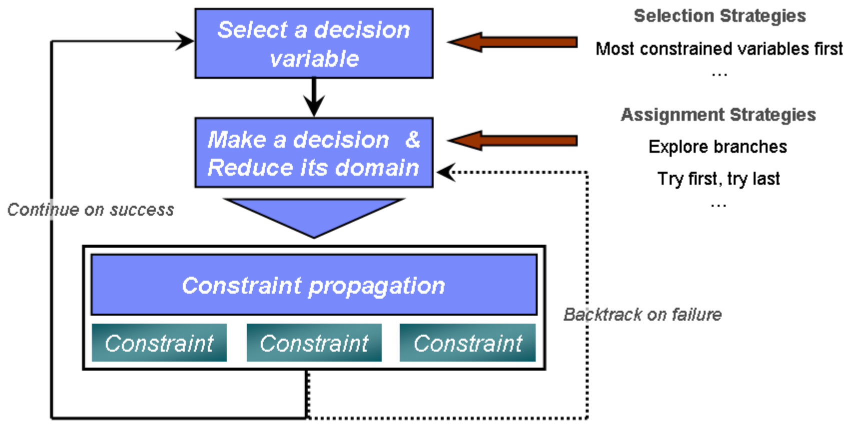
CP Optimizer uses a process called **constructive search** to *construct* a solution following these steps:

1. Select a decision variable
2. Assign a value to the decision variable
3. Reduce the domains of the other variables using constraint propagation
4. If step 3 fails, backtrack to step 2 and assign a different value to the variable, otherwise return to step 1

The process continues until all decision variables have a value, or it is established that no solution exists.

The following diagram illustrates the constructive search process:

CP constructive search process



Lesson Title

© Copyright IBM Corporation 2009

The **search space** is the product of all domain sizes, measured by its logarithm, and is a measure of how difficult a problem is for the CP Optimizer engine.

Search strategies

With CP Optimizer, the user does not need to describe a search strategy, because CP Optimizer will automatically generate a search based on:

- the model structure
- constraint propagation

However, the user can optionally fine-tune the search strategies by:

- modifying search parameters
- specifying more detailed search phases

CP search phases

In general, CP Optimizer's built in search works well without additional guidance.

However, a search phase can be used to tune a search strategy by specifying the criteria for the order in which decision variables are chosen to be fixed and/or to which values these variables should be fixed.

This strategy is then used to instantiate the decision variables of the phase.

Specifying a search phase can in some cases have a significant influence on the processing time.

Search phases can be composed of a subset of the following:

- an array of integers to instantiate (or fix)
- a variable chooser that defines how the next variable to instantiate is chosen

- a value chooser that defines how values are chosen when variables are instantiated

In the OPL model, search phases are written using IBM® ILOG® Script **`execute`** blocks located after the decision variable declarations and before the objective function definition.

A typical search phase definition, where **`x`** is a decision variable, is as follows:

```
execute {
 var f = cp.factory;
 var phase1 = f.searchPhase(x);
 cp.setSearchPhases(phase1);
}
```

Note that **`cp`** in the code above gives the user access to an instance of the **`IloCP`** class and its methods, and **`factory`** is a property of this class that accesses the CP search modifier factory. For more information, see the **IBM ILOG Script Reference Manual** available in the OPL Language manual.

You can also specify a sequence of search phases, for example:

```
execute {
  var f = cp.factory;
  var phase1 = f.searchPhase(x);
  var phase2 = f.searchPhase(y);
 cp.setSearchPhases(phase1, phase2);
}
```

This tells CP Optimizer to search first on the **`x`** decision variable and then on the **`y`** decision variable.

The general syntax of a phase that specifies both a variable chooser and a value chooser is as follows:

```
var phase1 = f.searchPhase(x,<variable chooser>, <value chooser>);
```

where **`<variable chooser>`** specifies how the next decision variable to fix in the search is chosen, and

`<value chooser>` specifies how values are chosen for instantiating decision variables.

A variable chooser is a combination of selectors and evaluators. For instance,

```
 f.selectSmallest(f.domainSize())
```

is a variable chooser that evaluates the domain size (the evaluator is **`f.domainSize()`**) of each variable and selects the one having the smallest size (the selector is **`f.selectSmallest()`**).

A value chooser is defined according to the same template. For instance,

```
f.selectLargest(f.value())
```

selects the largest value of the domain to instantiate the variable chosen.

The search phase using these choosers is then

```
var phase3 = f.searchPhase(z, f.selectSmallest(f.domainSize()),
f.selectLargest(f.value()));
```

Several predefined evaluators are available in the OPL documentation at **Language > Language User's Manual > IBM ILOG Script for OPL > Using IBM ILOG Script in constraint programming > Defining search phases**.

CP models in OPL

Learning objective
Learn the basics of how to use CP Optimizer and OPL to solve certain combinatorial optimization problems, other than scheduling problems.

Key terms
- arithmetic constraints
- logical constraints
- compatibility constraints
- specialized constraints

CP models for combinatorial optimization

In this section, you explore how to use CP Optimizer and OPL to model and solve certain combinatorial optimization problems other than scheduling problems. These are problems that are not well-suited for MP methods, but effectively solved using CP. The focus here is mainly on the syntax and basic modeling constructs.

The information discussed here is generally valid for any CP application in OPL. However, note that another lesson, **Scheduling in OPL with CP Optimizer** provides a thorough overview on using CP Optimizer and OPL for scheduling problems and syntax specific to scheduling problems is not covered in this lesson.

Invoking the CP Optimizer engine

The OPL keyword **`using`** at the beginning of a model file invokes the solution engine to be used. To call IBM ILOG CP Optimizer as the optimization engine, use the command:

```
using CP;
```

as the first line of your **`.mod`** file.

The **`using`** keyword can also invoke the CPLEX optimizer. In OPL V6.3, the CPLEX optimizer is used by default, and if no **`using`** command is present, CPLEX will be called. If a model file contains CP-specific constraints and **`using CP`** is not present, OPL will produce error messages.

Arithmetic operations, expressions and constraints in OPL

A CP modeler has access to a large number of arithmetic tools in OPL. These are shown in the table below and can be grouped as:

- Operations
- Expressions
- Constraints

While many of these are also available in MP, some are limited to CP in constraints, but can still be used outside constraint blocks with integer arrays for both CP and MP. The tools that fall in the latter category are indicated with a * in the **Explanation** column in the tables that follow.

Type of arithmetic tool	Operator	Representation in OPL	Explanation
Operations	addition	`+`	The usual arithmetic operations
	subtraction	`-`	
	multiplication	`*`	
	integer division	`div`	Returns the integer portion of a division, for example, `16 div 5 = 3`
	floating-point division	`/`	Returns the full floating-point result of a division, for example, `16 / 5 = 3.2`
	modular arithmetic	`%` or `mod`	In the expression `x % y`, returns the integer remainder of `x` divided by `y.` For example, `48 % 10 = 8`

Type of arithmetic tool	Operator	Representation in OPL	Explanation
Expressions (to be used in constraints)	Standard deviation	**standardDeviation <int_array>**	Returns the standard deviation of the **<int_array>**. **see note below*
	minimum	**min**	Aggregate operator that computes the minima of a collection of related expressions
	maximum	**max**	Aggregate operator that computes the maxima of a collection of related expressions
	count	**count (<arg_1>,<arg_2>)** where **<arg_1>** is an integer array and **<arg_2>** is an integer value (declared or calculated).	Counts how many of the elements in the array given as **<arg_1>** are equal to the value given as **<arg_2>.** **see note below*
	absolute value	**abs(float f)**	Returns the absolute value of **f**
	element or index	**element(<int_array>,n)** where **n** is an integer decision variable	Returns the **n**th element of **<int_array>.** **see note below*
Arithmetic constraints	equal to	**==**	The usual arithmetic evaluations. These are the same as are used in the C and C++ programming languages.
	not equal to	**!=**	
	strictly less than	**<**	
	strictly greater than	**>**	
	less than or equal to	**<=**	
	greater than or equal to	**>=**	

*Note that in constraints, these functions are limited to CP. Outside constraint blocks, they can be used with integer arrays both in CP and CPLEX.

Logical constraints in CP

IBM ILOG CP Optimizer provides full native support for logical constraints (for example **`and`**, **`or`** and **`not`**), and these logical operators are available in OPL.

Logical operator	Representation in OPL	Explanation
logical-and (conjunction)	**`<expression1> and <expression2>`**	Returns 1 (true) if both **`<expression1>`** and **`<expression2>`** are true, and 0 (false) otherwise.
logical-or (disjunction)	**`<expression1> or <expression2>`**	Returns 1 (true) if at least one of **`<expression1>`** or **`<expression2>`** are true, and 0 (false) otherwise.
logical not (negation)	**`!`**	Returns 1 (true) if the constraint is false, and 0 (false) otherwise.
logical if-then (implication)	**`if (<expression>) {<constraint(s)>}`**	If **`<expression>`** is true, include **`<constraint(s)>,`** otherwise, do nothing.
logical if-then-else (implication)	**`if (<expression>) {<constraint1(s)>} else {<constraint2(s)>}`**	If **`<expression>`** is true, include **`<constraint1(s)>,`** otherwise, include **`<constraint2(s)>.`**

The **`and`** and **`or`** operators allow you to express constraints in a more compact form by aggregating several constraints into a single expression, for example:

```
or(i in 1..5 : i mod 2 == 0) x[i] == 2;
and(i in 1..5 : i mod 2 == 1) x[i] == 1;
```

Compatibility Constraints

The following compatibility constraints are specific to CP, although they can also be used with integer arrays outside constraint blocks when using either CP or CPLEX models:

- **`allowedAssignments`**
- **`forbiddenAssignments`**

The purpose of these constraints is to define combinations of allowed or forbidden values for multiple integer decision variables.

These constraints can apply to any number of decision variables (and therefore each has a variable number of arguments). The set of allowed (or forbidden) combinations is given by a tuple with the number of fields equal to the number of considered decision variables. Each tuple defines an allowed (or forbidden) combination. Here is an example of their use:

```
using CP;

tuple C {
   int a;
   int b;
};

{C} possibles = {<1,1>, <2,4>};
{C} forbidden = {<3,5>};

dvar int+ x;
dvar int+ y;

constraints {

  allowedAssignments(possibles, x, y);

  forbiddenAssignments(forbidden, x, y);
}
```

Specialized constraints

A **specialized constraint** is equivalent to a set of arithmetic or logical constraints. They express complicated relations between decision variables that would otherwise require a large number of arithmetic constraints. Specialized constraints enter into such considerations as, for example:

- Counting values
- Maintaining load weights

In most cases, a specialized constraint achieves more domain reduction than the equivalent set of basic constraints, and in all cases it performs domain reduction more efficiently.

These constraints are described in the following table. All return Boolean values of 1 if the constraint is true, 0 otherwise:

Specialized constraint	Syntax example	Explanation
`allDifferent`	**`allDifferent(dvar int[ ])`**	Constrains decision variables within a **`dvar`** array to all take different values
`allMinDistance`	**`allMinDistance(dvar int[ ],x_int)`**	Takes two arguments: a **`dvar`** array and an integer value, **`x_int.`**Implies that values assigned to any two **`dvar`**s in the array differ by at least **`x_int.`**
`inverse`	**`inverse(dvar int[x],dvar int[y])`** where **`dvar int[x]`** and **`dvar int[y]`** are two one-dimensional arrays of integer decision variables that are indexed by an integer.	Returns a boolean value of 1 (true) if **`x`** and **`y`** are inverse functions, i.e. in the constraint **`inverse(x, y)`** for any value **`i`** of the indexer of **`y`** it is true that **`x[y[i]] = i`** otherwise, the result is 0 (false).

Specialized constraint	Syntax example	Explanation
`lex`	**`lex(dvar int[ ],dvar int[ ])`**	States that the values of the first array of decision variables is less than or equal to the values of the second array of decision variables, in lexical order. For example, the line **`lex (a,b)`** means that the value of **`a`** is smaller, or has lower order, in a lexical sense, than the value of **`b.`**
`pack`	**`pack(dvar int[ ],dvar int[ ],int[])`**	Maintains the load of a set of containers or bins, given a set of weighted items and an assignment of items to containers. See **OPL Functions > About the Language Quick Reference > pack** in the **Language Quick Reference** manual of the user documentation for a more complete explanation with examples.

In constraints, these functions are limited to CP. Outside constraint blocks, they can be used with integer arrays (not required to be **`dvar`** arrays) both in CP and CPLEX models.

The all quantifier

The quantifier **`all`** enables you to collect variables dynamically in an array. It functions similarly to the **`forall`** quantifier, but is used in a different part of the model.

When used together with one of the specialized constraints, it allows you to select part of an array according to the constraint applied. For example, the declaration

```
lex(all(i in 1..3) c[i], all(i in 4..6) c[i]);
```

means that the group of numbers composed of the first three digits of **`c`** comes before the group composed of the last 3 digits of **`c,`** in *lexical* (alphabetical) order.

The **`all`** quantifier preserves the order of the iteration when there is an index, and is very useful in conjunction with specialized constraints that use the order of the array of variables passed as arguments.

Comparison of all and forall

`forall` is a loop to create instances of constraints in a model.

`all` is a loop to create a set of objects that are used as arguments for certain CP constructs.

The following code illustrates this use of **`all:`**

```
-----------------------
using CP;

dvar int a[1..3] in 1..10;
dvar int b[1..3] in 1..10;

dvar int c[1..6] in 1..10;
```

```
constraints
{
  lex(a,b);

  allDifferent(append(a,b));

  lex(all(i in 1..3) c[i], all(i in 4..6) c[i]);
  allDifferent(c);
}
```

The results, in the **Solutions** tab of the **Output** window, are as follows:

```
CP found a solution:
a = [1 5 3];
b = [2 4 6];
c = [1 6 5 2 3 4];
```

Instructor note
The Steel Mill workshop shows how to use IBM ILOG CP Optimizer for a non-scheduling application. If the trainees are only interested in using IBM ILOG CP Optimizer for scheduling, you can skip the Steel Mill workshop and only do the workshops in the CP for Scheduling lesson. However, implementing a search phase is not covered in the Scheduling labs so if you want students to practice implementing a search phase, it's better to complete at least the first two steps of the steelmill lab.

Practice

The Steel mill problem
Go to the **Steel mill inventory matching** workshop and perform all the steps.

You can perform this lab using the HTML workshop, or by following the instructions in the workbook. The HTML workshop will give you direct access to OPL documentation pages that can help you with the lab.

Steel mill inventory matching

Workshop overview

In this workshop, you will practice using IBM® ILOG CP Optimizer to solve a steel mill inventory matching problem. We first present a description of the problem the production manager faces. Then you'll get a chance to derive the CP model from this description and model the problem using the OPL IDE. You'll also get a chance to experiment with search phases and alternative formulations to improve the solution speed.

Problem description

The steel mill has an inventory of steel slabs, of a finite number of different capacities (sizes), that are used to manufacture different types of steel coil. During production, some of the steel from the slabs is lost. The production manager has to decide which steel slabs to match with which coil orders in order to minimize the total loss. In optimization terms, the problem can be described as follows:

- ***Objective***
  - *Minimize the total loss*
- ***Decision variables***
  - *Which slab should be matched with which coil order*
  - *The capacity of each slab used*
- ***Constraints***
  - *A coil order can be built from at most one slab, although each slab can be used to fill several coil orders.*
  - *Each type of steel coil requires a specific production process, with each such process encoded by a color designated by a number between 1 and 88. A slab can be used for at most two different coil production processes or colors.*
  - *Each steel coil order has an associated weight, and the total weight of all coil orders matched with a slab must be less than the capacity of that slab.*
  - *The amount of loss from each slab equals the capacity of the slab minus the total weight of all coil orders assigned to that slab.*
  - *The total loss is the sum of losses from all slabs.*
  - *An unlimited quantity of steel slabs of each capacity is available.*

Problem data

*There are 111 coil orders and 21 different slab sizes, namely 12, 14, 17, 18, 19, 20, 23, 24, 25, 26, 27, 28, 29, 30, 32, 35, 39, 42, 43, and 44. The table below gives the **weight** and **color** data for 5 coil orders. The complete data set can be seen in the data file, which can be found in the exercise folder referred to below.*

Coil order	*Weight*	*Color*
1	*30*	*73*
2	*30*	*74*

Coil order	*Weight*	*Color*
3	*30*	*75*
110	*2*	*6*
111	*2*	*4*

Even though an unlimited quantity of steel slabs is available, you can use an upper bound of 111 on the total number of steel slabs because of the obvious solution of using one slab per coil order.

Exercise folder

`<trainingDir>\OPL63.labs\SteelMill\work`

The solution to this exercise is available in ***`<trainingDir>\OPL63.labs\SteelMill\solution`***

Solve the problem using CP Optimizer and the OPL IDE

Actions

- *Declare the data*
- *Declare the decision variables*
- *Define the objective function*
- *Define the constraints*
- *Solve the problem*

Reference

constraint programming

Declare the data

1. *In the OPL IDE, import the project by selecting* ***File >Import >Existing Projects into Workspace****, and choosing the* ***SteelMill_work*** *project from the exercise folder* ***`<trainingDir>\OPL63.labs\Steelmill\work`****. Leave the* ***Copy projects into workspace*** *box unchecked.*
2. *Expand the project to see the contents. For this step, you'll only work with the following:*
   - *Model file:* **`steelmill_work.mod`**
   - *Settings file:* **`steelmill_work.ops`**
   - *Data file:* **`steelmill_work.dat`**
   - *Run configuration:* **`naive model (default)`**
3. *Open the model file,* **`steelmill_work.mod`***, and see that the following data has been declared:*
   - *The number of coil orders:* **`int nbOrders = ...;`**
   - *The weight of each coil order:* **`int weight[1..nbOrders] = ...;`**

   Now use similar syntax to declare the following:

   - *The integer number of steel slabs available:* **`nbSlabs`**
   - *The integer number of colors:* **`nbColors`**
   - *The integer number of distinct slab capacities:* **`nbCap`**
   - *An integer array for the actual capacities associated with each index in* **`nbCap`***:* **`capacities`**

- *An integer array for the colors associated with each coil order:* **`colors`**

4. *The remaining completed items in the data declaration part of the model file are used later in the model, and are as follows:*
    - **`maxCap`**: *This is the greatest available capacity and will be used to define the domain of the decision variables for slab capacity.*
    - **`caps`**: *This is the set of available capacities (with the same content as the array* **`capacities`***) and is used in a later substep to define the capacity constraints for each slab.*
5. *Open the data file,* **`steelmill_dat.dat`***, to see how the data is instantiated.*

Declare the decision variables

1. *In the model file,* **`steelmill_work.mod`***, tell OPL that you're using CP by adding the line* **`using CP;`** *at the top.*
2. *In the section titled* **`/* Decision variables */`***, the two decision variables have already been declared as follows:*
    - **`dvar int where[1..nbOrders] in 1..nbSlabs`**: *This variable is used to determine which slab each coil order is assigned to, and is therefore indexed over all orders with domain of all slab numbers.*
    - **`dvar int capacity[1..nbSlabs] in 0..maxCap`**: *This variable is used to determine the capacity of each slab, and is therefore indexed over all slab numbers, with domain of all values between 0 and the maximum available capacity.*
3. *Declare an integer decision expression called* **`load`** *indexed over the slabs. Assign the value of* **`load`** *to equal the sum of the weights of all coil orders assigned to a slab.*

    First write the expression to sum the weights of all coil orders. Next, use the **`where`** *variable, together with the logical equals (==), to write an expression that evaluates to 1 if an order is assigned to a slab and 0 otherwise. Multiply this expression with the* **`weight`** *to add only the weight of orders assigned to the slab.*

4. *Declare another integer decision expression called* **`colorAssigned`** *indexed over the colors and the slabs. Complete the declaration with an expression to assign the value of* **`colorAssigned`** *to be 1 if any coil orders assigned to a given slab have a particular color and 0 otherwise.*

    First use the **`where`** *variable and the logical equals (==) that you used for the* **`load`** *expression to determine whether an order is assigned to a slab (1) or not (0). Next, in the same expression, use the logical OR statement (***`or`***) to create an "or" over all assigned orders with the particular color. Be sure to check the syntax for the logical OR statement in the online documentation at* ***Language Quick Reference > OPL keywords****.*

Define the objective function

1. *In the section titled* **`Objective function`***, write the objective to minimize total loss. The total loss is defined as the sum, over all slabs, of the slab capacity minus the slab load.*

Define the constraints

1. *In the section titled* **`Constraints`** *and within the* **`subject to`** *block, you'll see a* **`forall`** *constraint declaration that uses the* **`caps`** *set to*

restrict the domain of the **`capacity`** *variable. Because the* **`capacity`** *variable has a domain enumerated from a set, its domain cannot be defined during data declaration as follows:*

```
dvar int capacity[1..nbSlabs] in caps; // NOT ALLOWED
```

Instead, a continuous domain has to be defined at declaration, and the domain then has to be restricted in the constraint block.

2. *Add a constraint within this same* **`forall`** *block to state that the slab load must be less than or equal to the slab capacity.*
3. *Add a constraint called* **`colorCt`** *for each slab that states that the number of colors assigned to that slab must be less than or equal to 2.*

   Use the **`colorAssigned`** *decision expression.*

4. *Your model is complete at this point. If you have any remaining errors, check the solution or check with your instructor before attempting to solve the model.*

Solve the model

1. *In the* ***OPL Projects Navigator****, expand the* ***Run Configurations*** *for your project. right click* ***naive model (default)*** *and select* ***Run this*** *from the context menu.*
2. *Select the* ***Engine log*** *output tab to see the solution progress.*
3. *Let the model run for about a minute and then click the red stop button (you can see the solution time scrolling by periodically on the left side of the log).*
4. *Scroll to the start of the log and notice that the first solution found (under the* **`Best`** *column), is around 1000. Scroll to the end of the log and notice that the best solution found after about a minute is around 30. In the next step you'll get a chance to implement a search phase to improve performance.*

Implement a search phase

Actions

- *Add a search phase*
- *Solve the model*

Reference

what is a search phase?

Add a search phase

In this exercise you'll write a search phase to guide CP Optimizer in the selection of decision variables during the solution search. For the steel mill problem, an intuitive search strategy is to first assign orders to slabs (first search on the **`where`** *decision variable), and afterwards assign capacities to each slab (next search on the* **`capacity`** *decision variable).*

If you're confident that your model is correct, you can continue working with it. Otherwise, you can continue with the **`searchPhase_work.mod`** *file. These instructions assume you're using the latter file, which contains the model up to this point, together with some instructions on how to do implement the search phase.*

1. *Scroll down to the section titled* **`Search phase`**. *You'll write the search phase within the* **`execute`** *statement.*

2. *Define the script variable* **`f`** *to access the CP search modifier factory.*
3. *Define a search phase,* **`phase1`***, on the* **`where`** *variable using the* **`searchPhase`** *method.*
4. *Define another search phase,* **`phase2`***, on the capacity variable.*
5. *Use the* **`setSearchPhase`** *method to set the search to first use* **`phase1`***, and next* **`phase2`***.*
6. *If you have any errors, check the solution or check with your instructor before attempting to solve the model.*

Solve the model

1. *In the* ***OPL Projects Navigator****, expand the* ***Run Configurations*** *for your project. right click* ***search phase*** *and select* ***Run this*** *from the context menu.*
2. *Select the* ***Engine log*** *output tab to see the solution progress.*
3. *Let the model run for about a minute and then click the red stop button (you can see the solution time scrolling by periodically on the left side of the log).*
4. *Scroll to the start of the log and notice that the first solution found (under the* **`Best`** *column), is around 20 – much better than the first best solution of 1000 without the search phase. Scroll to the end of the log and notice that the best solution found after about a minute is around 8, again an improvement compared to the solution of 30 without a search phase. In the next step you'll get a chance to try and improve performance by changing the OPL model.*

Improve the model

Actions

- *Improve the model*
- *Solve the model*

Improve the model

The key to understanding the model improvements in this exercise, is realizing that once the load on a slab is known, its capacity becomes a trivial decision. Specifically, if the load is known, the capacity of that slab will simply be the smallest available capacity just bigger than the load on the slab. In this exercise, you'll take advantage of this knowledge to improve the model by removing the **`capacity`** *decision variable.*

1. *In the same work project in the OPL IDE, open the* **`betterModel_work.mod`** *file.*
2. *In the* **`Data declaration`** *section of the model file, notice that a new line has been added:*

   ```
   int loss[c in 0..maxCap] = min(i in 1..nbCap :
   capacities[i] >= c) capacities[i] - c;
   ```

   This line takes advantage of the fact that the load (and loss) is an integer and there are therefore a finite number of possible values the load on a slab can take, namely all integer values between **`0`** *and* **`maxCap`***. For each such value, the array above defines the loss to be the smallest capacity just larger than the load (defined by using the* **`min`** *function), minus the load. The array uses the index* **`c`** *and you'll see next how it can be used with the load values instead.*

3. *In the* **`Decision variables`** *section, remove the* **`capacity`** *decision variable. All the other variables remain the same.*

4. *Change your objective function to minimize the* **`loss`** *directly instead of using the* **`capacity`** *and* **`load`** *decision variables.*

   Index the **`loss`** *array with the* **`load`** *variable.*

   If you're familiar with MP, you'll notice here one of the major differences between MP modeling and CP modeling: In CP a decision variable can be used to index an array, as is shown here where the **`load`** *variable is used as an index for the* **`loss`** *array.*

5. *In the* **`Constraints`** *section, remove the constraints that use the* **`capacity`** *decision variable, seeing that they are no longer required when this variable doesn't exist.*
6. *Finally, in the* **`Search phase`** *section, remove* **`phase2`** *seeing that you no longer have the* **`capacity`** *variable.*
7. *If you have any errors, check the solution or check with your instructor before attempting to solve the model.*

Solve the model

1. *In the* ***OPL Projects Navigator****, expand the* ***Run Configurations*** *for your project. right click* ***Better Model*** *and select* ***Run this*** *from the context mneu.*
2. *Select the* ***Engine log*** *output tab to see the solution progress.*
3. *See that IBM ILOG CP Optimizer finds the optimal match between orders and slabs, resulting in zero loss, in about 0.1 seconds.*

   The steel mill problem is a benchmark problem used to benchmark optimization engine performance, and it's worth noting that IBM ILOG CP Optimizer is the first constraint programming engine to be able to find an optimal solution to this problem.

Use the pack constraint

Action

- *Use the* **`pack`** *constraint*

Reference

pack

Use the pack constraint

This part of the exercise is optional and shows you how to use one of IBM ILOG CP Optimizer's more advanced constructs, namely the **`pack`** *constraint. This constraint is generally used to assign items into packs of finite capacity. In this sense, the orders are the items, and the slabs are the packs of finite capacity to which the items are assigned (see the* ***OPL Help*** *for further information on the* **`pack`** *constraint).*

1. *In the same project you've been working in, open the* **`polished_work.mod`** *file.*
2. *In the* **`Data declaration`** *section, see that a new property has been declared, namely* **`maxLoad`***. This is the sum of the weights of all coil orders.*
3. *In the* **`Decision variables`** *section, see that* **`load`** *is no longer a decision expression, but is now an integer decision variable with domain*

between 0 and **`maxLoad`**. *The reason for this change is that the* **`pack`** *constraint does not accept a decision expression as an argument and instead requires a decision variable.*

4. *In the* **`Constraints`** *section, look at the definition of the* **`pack`** *constraint and try to understand it by comparing it with the explanation in* ***OPL Help***.
5. *Run the model from the* ***Polished Model Run Configuration*** *and see that it also finds the optimal solution of zero loss in around 0.1 seconds.*

Summary

Review

Constraint programming is a discipline of computer science based on logic and symbolic reasoning. It is very effective in solving, for example:

- detailed scheduling problems
- routing problems
- satisfiability problems
- certain other combinatorial optimization problems not well-suited for traditional Mathematical Programming (MP) methods

Some of the benefits of IBM ILOG CP Optimizer are as follows:

- **easily accessible through OPL**, taking full advantage of OPLs development, debugging and tuning features
- **automatically generates advanced algorithms** based on the mathematical formulation of a particular model
- allows users to "**model and run**" complex problems, without having to write the complex searches traditionally required for CP
- allows the advanced user to specify a search by using **search phases**
- **easy-to-use** representations for specialized **constraints**

CP methodology, in general, has two phases:

1. Write a model representation of a problem in a computer programming language
2. Describe a search strategy for solving the problem

A typical CP search uses the following techniques iteratively until a solution is found:

- Domain reduction: The process of eliminating possible values from a variable domain by considering the constraints on that variable and related variables.
- Constraint propagation: The process of communicating the domain reduction of a decision variable to all of the constraints on this variable. This can result in more domain reductions.

The command **`using CP`** tells OPL to use IBM ILOG CP Optimizer, and gives the user access to four types of constraints:

- Arithmetic
- Logical
- Compatibility
- Specialized

IBM ILOG CP Optimizer is IBM's second-generation CP engine and is distinguishable from IBM's first-generation CP engine called IBM ILOG CP. IBM ILOG CP remains available for complex routing problems that are beyond the current capabilities of IBM ILOG CP Optimizer.

Lesson 6: Infeasibility and Unboundedness - When the Problem Can't be Solved

What if there's no solution?

Sometimes, the combination of constraints and bounds in a problem is such that there is no feasible solution. For example, in a staffing problem, you have 3 employees, A, B and C. Your constraints include:

- A doesn't get along with B so they can't work together
- B doesn't get along with C so they can't work together
- The job requires two skills. A and C both have one of those skills, and B has the other.

This lesson shows you how to detect and resolve this type of situation in LP problems, using two CPLEX® techniques available in OPL:

- Conflict refinement
- Minimal relaxation

This lesson includes a review of the two methods, and practice with simple examples.

Instructor note
This lesson should last about 1 hour, including the practices.

Solving the infeasible model

Learning objective
Learn how to use OPL to find a minimal relaxation of an infeasible model

Key terms
- infeasibility
- unboundedness
- conflict
- minimal relaxation

Conflict refinement and relaxation

Infeasibility in an LP model can be caused by:

- The existence of at least two constraints in conflict – i.e. all possible solutions to one are excluded from being a solution to the other
- **Unbounded** constraints or an unbounded model that render the search for an optimal solution impossible

OPL provides two techniques in the CPLEX® optimizer engine to help you resolve such problems:

- Conflict refinement finds constraints that are in conflict.
- Relaxation suggests minimal changes in constraints that will render the model feasible, by *relaxing* one or more of the bounds defined in the constraints.

Conflicts

A conflict is a set of mutually contradictory constraints and/or bounds within a model. In other words, it is impossible for all the constraints to be true. A conflict is said to be a **minimal conflict** if it becomes feasible when any one constraint or bound is removed from the set.

This minimal conflict usually concerns a subset of the constraints in the full model, thus making it easier to analyze the source of infeasibilities. OPL's conflict refinement technology finds a minimal conflict for you.

In OPL, when a conflict occurs, it is displayed in the **Conflicts** tab. The information in this tab expresses the necessary change to make the model feasible: you must remove or modify at least one of the conflicting constraints.

Removal of a conflicting constraint makes the model feasible *with respect to this conflict only*. There may be other conflicts in the model. If this is the case, you may need to detect and refine those other conflicts as well to render your model feasible.

Minimal relaxation

A **relaxation** is a modified model where some of the restrictions on decision variables and/or constraints have been relaxed. For example:

- An integer decision variable could become a continuous decision variable
- A decision variable could have its bounds relaxed from [0,100] to [0,200]
- The righthand side of a range constraint, such as `2x + y <= 10` could be relaxed to `2x + y <= 12`.

The OPL relaxation search process uses the CPLEX method `feasOpt`. This method looks for a way to make the instance of the problem more flexible so that the it becomes feasible while keeping modifications to a minimum. In other words, the "Suggested Relaxation" message in the **Relaxations** output tab is the **sufficient minimal change** to make the model feasible.

By default, in OPL, if a model is infeasible, a feasible (not optimal) solution is found by minimizing the sum of all required relaxations. This is said to be a **minimal relaxation**. Other options can be set in the settings file to define how these minimal changes are measured as well as to enable an optimal solution to be subsequently found.

It is also possible to prohibit decision variable relaxation in the settings file. You will explore how to do this in the practice at the end of this lesson.

Limits on infeasibility analysis

- All variables are considered for infeasibility analysis by default.
- Only labeled constraints are candidates for infeasibility analysis (conflict refiner/**`feasOpt`**).
- Only ranged constraints are relaxable. Logical constraints are not.

• Infeasibility analysis applies only to models solved by the CPLEX engine. There is no support for conflicts and relaxations for models solved by the CP Optimizer engine.

Practice

Solving the infeasible model

Go to the **Pasta Production and Delivery** workshop and perform the first action, **Study the conflicts and suggested relaxations,** of the **Solve the infeasible model** step.

You can perform this lab using the HTML workshop, or by following the instructions in the workbook. The HTML workshop will give you direct access to OPL documentation pages that can help you with the lab.

Study the conflicts and suggested relaxations

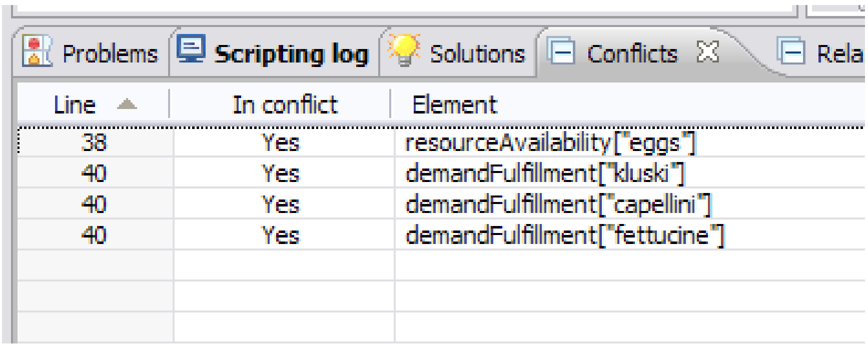
In this step of the workshop, you will be using the same model as the one you worked on in the previous step. For now, however, we have deliberately made this model infeasible by changing the value for the upper limit on outsourced production, the value of ***`maxOutsideProduction`****, from 200 to 50 in the data file.*

Procedure:

1. *Import the project* ***`<trainingDir>\OPL63.labs\Pasta\Tuples\solution\product2Work`*** *to the OPL Projects Navigator.*
2. *Run the model and observe the results in these* ***Output*** *window tabs:*
   - ***Engine Log***
   - ***Conflicts***
   - ***Relaxations***
   - ***Solutions***

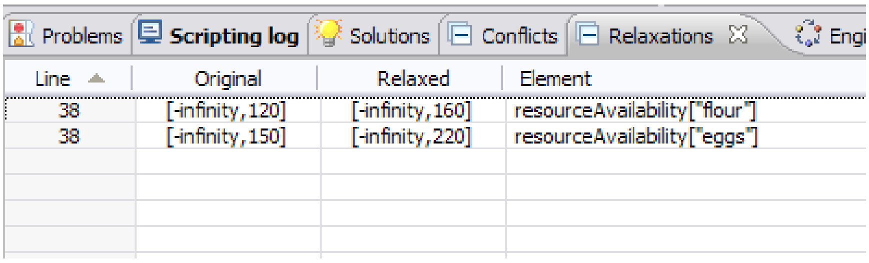
The ***Engine Log*** *tells you* ***`Implied bounds make row 'resourceAvailability ("flour")' infeasible`*** *- that is, there is no solution without relaxing at least one constraint.*

The ***Conflicts*** *tab tells you that there are conflicts between the* ***`resourceAvailability`*** *and* ***`demandFulfillment`*** *constraints.*



Line	In conflict	Element
38	Yes	resourceAvailability["eggs"]
40	Yes	demandFulfillment["kluski"]
40	Yes	demandFulfillment["capellini"]
40	Yes	demandFulfillment["fettucine"]

The ***Relaxations*** *tab suggests relaxations to the model in* ***`resourceAvailability`****.*



Line	Original	Relaxed	Element
38	[-infinity,120]	[-infinity,160]	resourceAvailability["flour"]
38	[-infinity,150]	[-infinity,220]	resourceAvailability["eggs"]

The proposed relaxation suggests that the supply of eggs be increased from 150 to 220. It also has calculated that if this is done, it will be necessary to increase the flour supply from 120 to 160.

Finally, the ***Solutions*** *tab calculates a "feasible relaxed sum of infeasibilities" i.e. a feasible, but not necessarily optimal, solution by minimizing the sum of all required relaxations given in the* ***Relaxations*** *tab. In this case, all the outside production is kept to the imposed limit of 50, and the internal capacity is increased to make up the difference in production internally.*

Before continuing to the next exercise, wait for the rest of the class and your instructor to discuss these results.

Which method to apply?

The practice problem presents 2 options for removing the infeasibility:

- Either relax the constraint, e.g. change **`availability`** to **`160`** for **`flour`**, and **`220`** for **`eggs`** *or*
- Remove the conflict by removing either the **`resourceAvailability`** or **`demandFulfillment`** constraint.

Which solution offers the best choice? To answer, it is important to know and understand your model:

- OPL reports the constraints involved in a conflict. You then need to determine which constraint is in error. The fact that a conflict exists for one constraint may, in fact, be the result of an error in modeling another constraint.
- Study what is reported in the **Conflicts** and **Relaxations** output tabs, remembering what each constraint and decision variable represents, and apply the solution that makes sense from a practical point of view.

Practice

In the **Pasta Production and Delivery** workshop, perform the second action, **Choose a method**, of the **Solve the infeasible model** step.

You can perform this lab using the HTML workshop, or by following the instructions in the workbook. The HTML workshop will give you direct access to OPL documentation pages that can help you with the lab.

Instructor note
Commenting out the **`resourceAvailability`** constraint produces a feasible solution in which no outside production takes place, as availability is no longer considered. While the solution is feasible, it is not correct from a real-world perspective, as there will not be enough resources to satisfy the production requirements. Discuss these results with the students.

Make the students aware that the relaxed solution suggested by OPL will not necessarily correspond to the solution obtained after applying the suggested relaxation to the original model. This is because when OPL attempts to find a relaxed solution, it uses a modified objective function defined by **`feasOpt`** in order to minimize the constraint violations required for a feasible solution, regardless of the original objective. After the user then “fixes” the infeasibilities by applying the suggested relaxation, the model goes back to using the original objective and therefore may find a different solution.

Choose a method

*Look at the suggested option(s) for solving the problem you have just run in the **Conflicts** and **Relaxations** output tabs.*

1. *Record the data from the **Solutions** tab for later reference.*
2. *Try commenting out the **`resourceAvailability`** constraint and run the problem again. Is the conflict resolved? Is the result meaningful?*
3. *Restore the **`resourceAvailability`** constraint and modify the data file as suggested by the **Relaxations** output tab - i.e. set*

   ```
   availability = [ 160, 220 ];
   ```

4. *Run the problem again.*
5. *Compare the result in the **Solutions** tab to the earlier results you recorded. These solutions are equivalent. Can you think of circumstances in which they would be different?*

Practice

Modify the feasOpt parameters

You can also modify what OPL takes into account when looking for a relaxation:

- Labeled constraints only
- All decision variables and all labeled constraints

Go to the **Pasta Production and Delivery** workshop and perform the third action, **Modify the feasOpt parameters,** of the **Solve the infeasible model** step.

Modify the relaxation level

Procedure:

1. *In the settings file, select* ***Language> General****.*
2. *In the* ***Relaxation Level*** *item, change the option in the drop-down list.*
3. *Solve the model again, and look at the information displayed in the output tabs.*

Instructor note
The **MP Staffing Problem** workshop includes a lab step, **Constraint relaxation**, which can be added to the above lab, *only after performing the first step of that lab, which requires database reading.* If you wish to use this exercise, schedule it after the database lesson, either by moving the database lesson earlier or this lesson later.

Summary

Review

In this lesson, you learned about the two methods provided in OPL to handle infeasibility:

- Conflict refinement seeks to identify the minimal number of constraints involved in a conflict. If removing one constraint or bound will make the model feasible, the conflict is said to be a minimal conflict.
- A relaxation is a modified model where some restriction on decision variables and/or constraints have been relaxed. The method **`feasOpt`** looks for a way to make the model and its data more flexible so that the problem becomes feasible while keeping modifications to a minimum.
- You can change how **`feasOpt`** defines a minimal relaxation in the settings file.

Lesson 7: Data Consistency

Data consistency is concerned with the accuracy and validity of data elements and inconsistency arises frequently as users are free to add, delete and modify data. If the model data is inconsistent the optimization solver will produce an incorrect solution or no solution. OPL offers these ways to check for data consistency in your projects:

- The **`with`** keyword (data membership consistency)
- The **`assert`** keyword (rules for maintaining data consistency)

The keyword **`key,`** dealt with earlier in this training in the context of tuples, also provides data consistency.

Instructor note
This lesson should last about 30 minutes.

Data membership consistency

Learning objective
Improve the efficiency of OPL models

Key term
with (OPL keyword)

The “with” keyword

The keyword **with** enables you to indicate that a given element of a tuple must be contained in a given set. If you use it, OPL checks the consistency of the tuple set at run time when initializing the set. The syntax is:

```
{tupletype} tupleset with cell1 in set1, cell2 in
set2 = ...;
```

For example, you have a set of arcs between nodes. Nodes are defined by a tuple set of tuples consisting of an origin node and a destination node. The **with** syntax enables you to ensure that the origin and destination nodes belong to a specific set of nodes.

The following code:

```
{int} nodes = {1, 5, 7}
tuple arc  {
   int origin;
   int destination;
}
{arc} arcs2 with origin in nodes, destination in nodes =
   {<1,4>, <5,7>}
execute {
   writeln(arcs2);
};
```

will raise an error when the set **arcs2** is initialized.

This is because the **with** syntax will detect that the statement

```
(int) nodes = (1, 5, 7);
```

is not consistent with the statement

```
with origin in nodes, destination in nodes = {<1,4>, <5,7>}
```

Changing the last statement as follows:

```
with origin in nodes, destination in nodes = {<1,5>, <5,7>}
```

will make the model function properly.

Verifying data consistency

Learning objective
Improve the efficiency of OPL models

Key term
assert (OPL keyword)

Assertions

OPL provides assertions to verify the consistency of the model data. This functionality enables you to avoid wrong results due to incorrect input data. In their simplest form, assertions are simply Boolean expressions that must be true; otherwise, they raise an execution error. Example:

```
int demand[Customers] = ...;
int supply[Suppliers] = ...;

assert sum(s in Suppliers) supply[s] == sum(c in
Customers) demand[c];
```

This code makes sure that the total supply available from **`Suppliers`** meets the total demand of the **`Customers`**. If this assertion is found not to be true, an error is raised.

Using the “assert” keyword

Multiple combinations (of suppliers and customers, or of other decision variables) can be verified using **`assert`** in combination with the **`forall`** aggregator:

```
int demand[Customers] [Products] = ...;
int supply[Suppliers] [Products] = ...;
assert
   forall(p in Products)
 sum(s in Suppliers) supply[s][p] == sum(c in Customers)
demand[c][p];
```

This code verifies that the total supply meets the total demand for each product. The use of assertions makes early detection of data input errors possible, and avoids tedious inspection of the model data and results.

Summary

Review

In this lesson, you learned how to maintain the accuracy and validity of data elements in OPL models:

- Use the keyword **`with`** to indicate that a given element of a tuple must be contained in a given set. If you use it, OPL checks the consistency of the tuple set at run time when initializing the set.
- Use the keyword **`assert`** to ensure that the data used by the model meets the necessary conditions. If the assertion fails, an error is raised.

Lesson 8: Linking to Spreadsheets and Databases with OPL

Instructor note
This lesson should last about 1 hour and 30 minutes including the practice.

To help you make better business decisions, OPL is able to interact with existing tools that you may already have in place, such as spreadsheets and relational databases. Thus you can use and update your existing data automatically by telling OPL to communicate with your spreadsheet or database. This is especially important for large databases, for example a Human Resources employee information database.

In this lesson you will learn how to use OPL to read from, manage and update spreadsheets and databases. With these operations, OPL can:

- use information in an external application to initialize data in a model
- write the results of an optimization to new or existing locations in the external application
- delete database records or fields that are rendered obsolete by the optimization operation

This lesson contains references to IBM® ILOG® Script. You should be familiar with IBM ILOG Script for OPL before working with that part of this lesson.

Instructor note
The following OPL keywords are no longer valid:

- **`DBconnection`**
- **`DBread`**
- **`DBupdate`**
- **`DBexecute`**

You need to use the following modified syntax (with upper case letters) instead:

- **`DBConnection`**
- **`DBRead`**
- **`DBUpdate`**
- **`DBExecute`**

Exchanging data with a spreadsheet

Learning objective
Learn to use OPL to read from and write to a spreadsheet

Key terms
- SheetConnection
- SheetRead
- SheetWrite

To exchange data with a spreadsheet, you need to:

1. Establish a connection between the OPL application and the spreadsheet using **`SheetConnection`**
   - **`SheetConnection`** takes a handle – OPL will use this as the "pipeline."
2. Read data from the spreadsheet using **`SheetRead`**.
   - You can read from one or more sheets in an Excel file.
   - You can define a range of cells to read.
   - You can use an Excel named range.
3. Write data to the spreadsheet using **`SheetWrite`**.
   - You can write to one or more sheets in an Excel file.
   - You can define a range of cells to write to.
   - You can use an Excel named range.

Connecting to a spreadsheet

Before an OPL model can read from and/or write to a spreadsheet, it must connect to the spreadsheet using the **`SheetConnection`** instruction.

This is not done in the **`.mod`** file but in the **`.dat`** file.

Steps:

1. Write your model in the **`.mod`** file exactly as you would any model in a project with independent data.
2. To initialize the data, use links to the spreadsheet in the **`.dat`** file.

Syntax:

```
SheetConnection <handle> ("filename.xls");
```

Example:

```
SheetConnection connex ("mySheet.xls");
```

Establishes a connection **`connex`** to a spreadsheet named **`mySheet.xls`** in read/write mode.

Relative and absolute paths are both supported. Relative paths are resolved using the current directory of the **`.dat`** file.

Reading from a spreadsheet

Once the model is connected to the spreadsheet, data can be read into these data elements using the **`SheetRead`** command:

- One-dimensional arrays
- Two-dimensional arrays or sets

Steps:

1. In the **`.mod`** file, declare the array or set, **`<dataElement>,`** that you want to fill with data from the spreadsheet;
2. In the **`.dat`** file, call the data to be read using the syntax: **`<dataElement> from SheetRead (<handle>, "<SheetName>!<startCell>:<endCell>";`**

   `<SheetName>` is optional. If no **`<SheetName>`** is given, and the Excel workbook has more than one sheet, the currently active sheet is read.

Example:

`Streets from SheetRead (connex,”addresses!A1:C13”)`

Reads the range of cells from **`A1`** to **`C13`** into a data element named **`Streets`** from the sheet named **`addresses`** that is located in the spreadsheet that uses the **`connex`** connection.

where:

- **`<dataElement>`** is the array or set, declared in the model file, that you want to fill
- **`<SheetName>`** is the name of the sheet inside the Excel file that you want to read from.
- **`<startCell>`** is the first cell, in the spreadsheet called by the connection **`<handle>`**, of the range that you want to use to fill **`<dataElement>`**.
- **`<endCell>`** is the last cell, in the spreadsheet called by the connection **`<handle>`**, of the range that you want to use to fill **`<dataElement>`**.

Example for arrays:

```
/* in .mod file */
 string city[0..2] = …;
/* in .dat file */
  city from SheetRead(connex,"C2:E2");
/* in .mod file */
  float cost[0..2][0..2] = …;
/* in .dat file */
  cost from SheetRead(connex,"C3:E5");
```

Example for sets:

```
/* in .mod file */
  {string} Cities = …;
/* in .dat file */
  Cities from SheetRead(connex,”addresses!C2:E2");
```

If any data change is made to a spreadsheet, the spreadsheet must be saved in order to allow the OPL model to take new values into account.

Writing to a spreadsheet

You can write into spreadsheets using the **`SheetWrite`** command. As with **`SheetRead`**, you write the **`SheetWrite`** command in the **`.dat`** file.

Steps:

1. You must already have an array or set declared in the **`.mod`** file and instantiated, from which you will write to the spreadsheet. This is represented here by the token **`<dataElement>`**.
2. Try to figure out the syntax for **`SheetWrite`** and give an example of how it might be used.

Instructor note
Solution:

In the **`.dat`** file, write the data to a range in the spreadsheet, using the syntax:
`<dataElement> to SheetWrite (<handle>, “<SheetName>!<startCell>:<endCell>”;`

where:

- **`<dataElement>`** is the array or set, declared in the model file, that you want to fill
- **`<SheetName>`** is the name of the sheet inside the Excel file that you want to read from.
- **`<startCell>`** is the first cell, in the spreadsheet called by the connection **`<handle>`**, of the range that you want to use to fill **`<dataElement>`.**
- **`<endCell>`** is the last cell, in the spreadsheet called by the connection **`<handle>`**, of the range that you want to use to fill **`<dataElement>`.**

Example:

```
/* in .mod file */
  float cost[0..2][0..2] = ...;
/* in .dat file */
  cost to SheetWrite(connex,"widgets!G3:I5");
```

Using Excel named ranges

You can use an Excel named range instead of an absolute range of cells to define the area of the spreadsheet you want to read from or write to.

Steps:

1. Define a named range, **`<rangeName>`** in your spreadsheet.
2. In the **`.dat`** file, use a **`SheetRead`** or **`SheetWrite`** statement with the following syntax modification:

   ```
   <dataElement> from SheetRead (<handle>, "<rangeName>";
   ```

3. Write an example for both **`SheetRead`** and **`SheetWrite`** using named ranges.

Instructor note

Here are a couple, based on the previous examples:

```
Cities from SheetRead(connex,"majorUrbans");
cost to SheetWrite(connex,"widgetCost");
```

where **`majorUrbans`** and **`widgetCost`** are named ranges in an Excel spreadsheet.

How to define a named range in Excel:

In an **`.xls`** file:

1. Highlight the range values (i.e., cells within the rows and columns to be included in the range).
2. From the main menu, select **Insert >Name >Define**.
3. Enter the desired name for the range.
4. Click **OK**.
5. Save the **`.xls`** file.

Practice

Can you think of how using named ranges can simplify certain repeated operations in a spreadsheet, especially where the location of a named range of cells might change?

Instructor note

Take, for example, the following **`SheetRead`** call:

`SheetRead (manufacturing,"Sheet1!A1:A5")`.

In this example, **`manufacturing`** is the connection handle, and the range is **`"Sheet1!A1:A5"`**. You could replace this absolute range with a named range, for example, **`“Product”`**.

The named range is the better option, since you only need to update the Excel worksheet to maintain data integrity.

For example, if you delete the cells from **`Sheet1!A1:E5`** and paste them onto **`Sheet2`**, Excel updates your named range (**`Product`**). You don't need to modify the OPL model because Excel find the cells defined as the range, **`“Product”`**, wherever they may be found in the spreadsheet:
`SheetRead (manufacturing,"Product")`

Practice

Using SheetWrite when the range is only known at runtime

Suppose you are optimizing a transportation model, in which you will only know the number of shipments you will send out after solving the model. Let's also assume that you need to write the shipment solve results to a spreadsheet.

The model includes a tuple describing each shipment by points of origin and destination, and the total volume contained in the shipment:

```
tuple shipment {
  key string   origin;
  key string   destination;
  int        totalVolume;
}
{shipment} Shipments = ...;
```

As is often the case, the shipments are instantiated via a tuple set, in this case called **`Shipments`**.

You need to write the shipment information (members of the tuple **`shipment`**) to three columns (A-C) of a worksheet named **ScheduledTrips** in an Excel file named **InputData.xls**. Each row represents one shipment, the columns represent the shipment data (origin, destination, volume).

How can you specify the range?

Think about postprocessing.

Instructor note
You solve this using IBM ILOG Script to generate the Excel range, and then passing the result to the **`SheetWrite`** command. The solution is shown, step by step, in a series of slides that are in your slide deck, but are not reproduced in the workbook.

Students should take notes on this as you go through it, but first, try to elicit ideas from them as to how it could be done. If someone seems to have a pretty clear idea of how to go about it, let that person lead the group through the process, and just provide coaching when necessary.

Another possible way to do this is to treat the spreadsheet as an ODBC data source. Connect using the ODBC database syntax, and then update with a **`DBUpdate`** instruction to write the data.

Connecting to a database

Learning objective
How to connect an OPL model to an RDBMS

Key term
DBConnection

As with spreadsheets, it is possible to connect an OPL model to a database. IBM® ILOG® OPL interfaces with:

- DB2
- MS SQL
- ODBC
- Oracle 9 and later
- OLE DB

Connection syntax

`DBConnection <DBclient> <ConnectionSting>`

The OPL keyword **`DBConnection`** establishes a named connection to a database. It requires two arguments:

- The name of the database client you want to use
- The connection string

The first argument is one of the supported databases. The connection string passed as second argument must respect a format that depends on the target RDBMS. The following table lists database names (first argument) and connection strings (second argument):

Database client name	Connection string
DB2	username/password/database (The client configuration will find the server.)
MS SQL	userName/password/database/dbServer
ODBC	dataSourceName/userName/password
Oracle 9 and later	userName/password@dbInstance
OLE DB	<user>/<password>/<database name>/<server name>

Instructor note
You can explain more about certain databases if students need it. For example, there are some special requirements for Oracle, and if the student has MS Access installed, there is the special connection string described next that will supply additional functionality. Refer to the documentation.

For example, the instruction

```
DBConnection DBconnex ("ODBC","DBPEOPLE/user/password");
```

establishes a connection named **`DBconnex`** to an ODBC data source whose name is **`"DBPEOPLE"`**. The connection **`DBconnex`** should be viewed as a handle on the database.

The user and password are optional. To connect to a database without user and password, terminate the database name with two slashes to indicate that user and password are empty:
`DBConnection DBconnex ("ODBC","DBPEOPLE//");`

Special case for Microsoft Access

Microsoft® Access uses ODBC. A connection string for Access is provided to facilitate the use of this common database tool:

```
DBConnection db(“access” ,”afile.mdb”);
```

where **afile.mdb** is the name of an Access database file to which you want to connect.

OPL supports both Microsoft Access 2003 and Microsoft Office Access 2007 through the Office 2007 drivers. If you do not have the Office 2007 drivers installed, you will receive an error message such as

```
[Microsoft][ODBC Microsoft Access Driver] Cannot open database
 '(unknown)'.
```

You need to install the Office 2007 drivers on your machine even if you are using Access 2003 databases. You can download these drivers from the Microsoft website:
http://www.microsoft.com/downloads/details.aspx?FamilyID=7554F536-8C28-4598-9B72-EF94E038C891&displaylang=en

It is possible to connect to several databases within the same model. Once connected, available database operations are:

- **DBRead**
- **DBExecute**
- **DBUpdate**

DBConnection and **DBRead** are executed during preprocessing. **DBExecute** and **DBUpdate** are executed during postprocessing.

Preprocessing and **postprocessing** are notions in IBM ILOG Script, which is dealt with in another lesson. Preprocessing permits you to work on data before the optimization model is created by using IBM ILOG Script/JavaScript™ syntax encapsulated in **execute** blocks. Postprocessing allows you to use script commands to manipulate solutions, also using an **execute** block.

Reading from a database

Learning objective
Learn how to read data from an RDBMS into an OPL model using standard database queries and the OPL database connection

Key term
DBRead

In OPL, data can be read into sets using **`DBRead.`** The resulting set must be a set of integers, of floats, of strings, or a set of tuples whose elements are integers, floats, or strings.

Syntax

`<entity> from DBRead (<Id>, "Query");`

where:

- **`<entity>`** represents the OPL name for the data to be filled from the database
- **`<Id>`** represents the name of the database
- **`Query`** represents a valid query syntax for the database **`<Id>`**.

OPL does not parse the query; it simply sends the string to the database system that has full responsibility for handling it.

Example – method using temporary set

In earlier versions of OPL, arrays could not be filled directly from a database and a temporary set was required to facilitate this operation. Some types of arrays can now be filled directly from a database using a feature called **table loading.** The original method is useful if you need to generate more complex arrays, such as arrays of sets or multidimensional arrays, or if you'd like to use the same set to populate several different sets or arrays.

The examples that follow show the original method using a temporary set, and the new table loading method for reading directly from the database.

Consider the following extract from the model file, **`<OPLhome>\examples\opl\oilDB.prj`**:

```
{string} Gasolines = ...;
{string} Oils = ...;

tuple gasType {
   string name;
   float demand;
   float price;
   float octane;
   float lead;
}

tuple oilType {
   string name;
   float capacity;
   float price;
   float octane;
   float lead;
}
{gasType} GasData = ...;
{oilType} OilData = ...;
```

coupled with, in the data file:

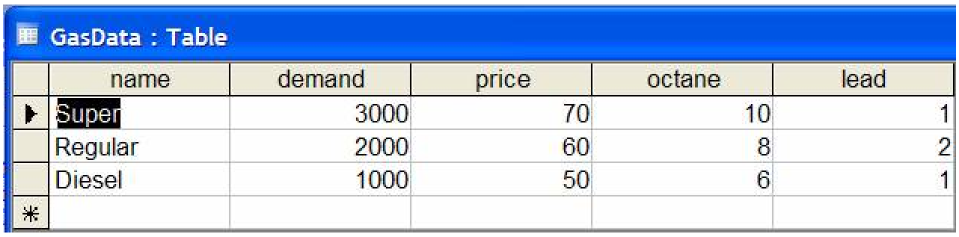
```
GasData from DBRead(db,"SELECT * FROM GasData");
OilData from DBRead(db,"SELECT * FROM OilData");
```

In this code, **GasData** and **OilData** are temporary sets that are initialized from the tables **GasData** and **OilData** in the **db** database. The **DBRead** instruction inserts a tuple into each set (**GasData, OilData**) for each row of the associated table. These sets are used later in the model to fill the one-dimensional **Gas** and **Oil** arrays, using the following code:

```
gasType Gas[Gasolines] = [ g.name : g | g in GasData ];
oilType Oil[Oils] = [ o.name : o | o in OilData ];
```

In the code above, the text to the left of the colon, **g.name**, indicates the index to use for the array (in this case the content of the set **Gasolines**), and the text to the right of the colon, **g | g in GasData**, indicates the corresponding array element. Note that the sets of strings, **Gasolines** and **Oils**, are filled at the same time as the array.

The columns of the SQL query result are mapped to the fields of the OPL tuples by position. For instance, in the above query that reads data from the **GasData** table below, the column **name** in the database table is mapped to the field **name** in the tuple, and so on. Note that the names of the database column and tuple field in general do not need to be the same, although it makes following the code easier.


GasData : Table

name	demand	price	octane	lead
Super	3000	70	10	1
Regular	2000	60	8	2
Diesel	1000	50	6	1

Example – method using table loading

One-dimensional arrays of simple data types (such as float, int or string) can be filled directly from the database, without going through the temporary sets (**GasData** and **OilData** in this example). Using this table loading feature, the model file looks like this:

```
{string} Gasolines = ...;
{string} Oils = ...;

tuple gasType {
   string name;
   float demand;
   float price;
   float octane;
   float lead;
}

tuple oilType {
   string name;
   float capacity;
   float price;
   float octane;
   float lead;
}

gasType Gas[Gasolines] = ...;
oilType Oil[Oils] = ...;
```

You can see that the tuple sets are no longer needed, and the one-dimensional arrays **`Gas`** and **`Oil`** are initialized directly from the data file. The new data file code is as follows:

```
Gasolines,Gas from DBRead(db,"SELECT
name,name,demand,price,octane,lead FROM GasData");
Oils,Oil from DBRead(db,"SELECT name,name,capacity,price,octane,lead
 FROM OilData");
```

The table loading form of **`DBRead`** is used to initialize the arrays directly. This is not only easier to write, it executes more efficiently.

SQL queries

Every valid SQL query is valid in OPL, including parameterized queries.

For example, declare a tuple type:

```
tuple People {
        string name;
        string email;
        int age;
   }
```

Declare an **`int`** parameter:

```
int age = 30;
```

Read tuples where **AGE** is smaller than 30:

```
{people} Persons = …;
Persons from DBRead (DB,"select NAME ,EMAIL,AGE from PEOPLE where
AGE<=?")(age);
```

The query contains a placeholder whose value is given by an expression in between the parentheses.

It is possible to use several placeholders. For example, with the same tuple declared above, declare two parameters:

```
int age = 30;
string dept = "marketing";
```

Read tuples matching two conditions:

```
{people} Persons = …;
Persons from DBRead (DB,"select NAME ,EMAIL,AGE from PEOPLE where
AGE<=? and DEPT=?")(age,dept);
```

This selects only those persons not older than 30 in the marketing department.

Writing to a database

Learning objective
Learn to write, update and delete data in an RDBMS using optimization results.

Key terms
- DBExecute
- DBUpdate

Two types of write operation are available:

- A direct write to a database updates it with values specified in the write command. It is similar to a database read.
- Publishing results to a database is similar to data initialization with parameters.
  All database publishing requests are carried out during postprocessing, if a solution is available. Such requests are processed in the order declared in the **`.dat`** file(s). If your RDMBS supports transactions, every single publishing request is sent within its own transaction.

You perform a direct write using the **`DBExecute`** statement.

The publishing process has two steps:

1. Use the **`DBExecute`** statement in the data file to create the table that will hold the results of the model solution in the database so that the database can be updated.
2. After a **`DBExecute`** instruction in a data file, the instruction **`DBUpdate`** calls the results to be published and modifies the data in the database using the result information.

DBExecute

`DBExecute` can be used to perform any database administration command, for example, writing data directly or creating a table. The syntax of **`DBExecute`** is:

```
DBExecute (<connection_name>, <command_string>)
```

where **`<connection_name>`** is the name given to the database connection when **`dbConnection`** was issued, and **`<command_string>`** identifies the action (e.g. SQL command) to be performed and its values.

Example – table creation:

```
DBExecute(db, "create table PERSONS(NAME string,EMAIL string,AGE
integer, DEPT string)");'
```

This code creates a table in the database **db**, with the heading **PERSONS.** The table has 4 columns, 3 headed **NAME**, **EMAIL** and **DEPT**, which are to be filled with string values, and one column headed **AGE**, which is to be filled with integer values.

Example - direct write:

```
DBExecute(db, "insert into PERSONS values (Mary, mary@company.com,
 34, Engineering)");
```

This code inserts a record in the table **PERSONS** with the following values:

- **NAME** = Mary
- **EMAIL** = mary@company.com
- **AGE** = 34
- **DEPT** = Engineering

DBExecute can be used in this fashion to perform any legal operation for the RDBMS to which you are connected (deleting a record, for example).

DBUpdate

The **`DBUpdate`** instruction is used to iteratively publish a set of results to the database. When it is invoked, the OPL result publisher will iterate on the items in the set result and bind the component values to the SQL statement parameters in the declared order. The element types supported for database publishing are the same as for reading. The syntax is as follows:

```
<data_instance> to DBUpdate (<connection_name> <command_string>)
```

where **`<command_string>`** represents the update action (an SQL statement, for example) and **`<data_instance>`** represents the OPL results (e.g. data or decision variable values) to which the action applies.

For example, after the **`DBExecute`** example that creates the **PERSONS** table shown above, you can specify, in the data file:

```
people to DBUpdate(db,"insert into PERSONS(NAME,EMAIL,AGE,DEPT)
values(?,?,?,?)");
```

This code, preceded by the **`DBExecute`** operation creating the table, inserts new records that fill the **PERSONS** table in the **`db`** database with information from a set, **`people,`** defined in the **`.mod`** file.

In the case of result publishing, **`DBExecute`** is not used to provide the values to fill, it simply performs the administrative operation (in this case, creating the table). The operation must be one which is legal for the type of RDBMS to which you are connected.

You can also use the **`DBUpdate`** instruction to update or delete existing database elements, using the appropriate SQL command. The following example deletes records, identified by the **NAME** column, and contained in the set **`namesToDelete:`**

```
{string} namesToDelete = ...;
namesToDelete to DBUpdate(DB,"delete from PERSONS where NAME = ?")
```

Practice

Go to the **Staffing Problem** workshop:

1. Read thoroughly the introductory material, especially the **Problem description**, which is not repeated in this workbook.
2. Perform the step, **Steps to the database solution.** Note that the instructions are repeated below.

Supply side data elements: Concept of skillGroup

Each worker belongs to a single **`skillGroup`** that represents a subset of all the skills of all the workers. In human resources terms, this would be the set of competencies for which a worker is qualified. An individual worker can share his/her available time performing any of the associated **`skillGroup`**'s skills, but cannot work in any skill outside the **`skillGroup`**.

Now perform the **Declare and instantiate worker and skillGroup pairs and worker's availability** substep of the **Steps to the database solution.** step.

You can perform this lab using the HTML workshop, or by following the instructions in the workbook. The HTML workshop will give you direct access to OPL documentation pages that can help you with the lab.

Declare and instantiate worker and skillGroup pairs and worker's availability

Each worker belongs to a single skillGroup and can share his available time across his skillGroup's skills.

In the model file:

1. *Look at the declaration (already done) of the tuple called **`WorkerSkillGroupPair`** that it contains the information for the **`skillGroupName`** and the **`workerName`** . Keys are used here for data integrity. Note that if no keys are declared, OPL's default setting is to assume that all tuple elements are keys.*
2. *Declare a set **`workerSkillGroupPairs`** of type **`WorkerSkillGroupPair`**.*

   *What you are doing here is creating a set, **`workerSkillGroupPairs`**, which has as its members, instances of the tuple **`WorkerSkillGroupPair`**. The tuple name is usually declared in the singular (**`WorkerSkillGroupPair`**), and the tuple set that collects it usually has the same name as the tuple data type but in the plural form (**`workerSkillGroupPairs`**). Also, note that usually, the data name starts with lower case while the data type starts with upper case (e.g. **`tuple Pair {...};`** and **`{Pair} pairs = ...;`**).*

3. *Declare an array **`workerAvailability`** of type **`float`** indexed over **`workerSkillGroupPairs`***

   *This array is a sparse array because it is indexed over the skill group name combination, **`WorkerSkillGroupPairs`** as opposed to being indexed individually over **`skillGroupName`** and **`workerName`**.*

In the data file:

1. *Populate **`workerSkillGroupPairs`** and **`workeravailability`** from the database using table loading. Use the following query:*

```
select sa.name,sg.skill_group_name, sa.availability from
skill_groups sg, skill_availability sa where
sg.skill_group_id = sa.skill_group_id;
```

Note that the index of the availability array and the elements of the array are created simultaneously from the database thanks to table loading.

2. *Use* **`workerSkillGroupPairs`** *to populate other data structures*
3. *Create the set of* **`namesOfWorkers`**

Data: the demand side

Demand is skill dependent. You therefore have to design several different structures to manipulate different aspects of the relationship between a worker and its associated skill set.

Now perform the rest of the substeps in the **Steps to the database solution.** step.

Declare and instantiate the skills list and the demand for each skill

In the model file:

1. *Declare the skills using a set structure*
2. *Declare an array of the skill's demand indexed by each skill name*

In the data file:

- *Populate the skill set and the demand array from the database using table loading. Use the following query:*

```
select sa.name, s.skill_name from skill_availability
sa, skills s, skill_group_skills sgs where
sa.skill_group_id = sgs.skill_group_id and sgs.skill_id
= s.skill_id;
```

Data: get relation of skills to workers

Each worker provides several skills, which are included in the pool of skills of his skill group

In the model file:

1. *The model file already contains the declaration of a tuple* **`WorkerSkillPair`** *that contains a worker and one of the worker's associated skills.*
2. *Declare a set* **`workerSkillPairs`** *of type* **`WorkerSkillPair`** *that is read from the data base.*

   What you are doing here is creating a set, **`workerSkillPairs`**, *which has as its members, instances of* **`WorkerSkillPair`**.

3. *Use the set* **`workerSkillPairs`** *to create an array* **`workerSkillsList`**, *indexed over* **`namesOfWorkers`**, *that lists the skills belonging to each worker.*

   Here, you are creating a two-dimensional array where one dimension is the name of each worker and the other dimension contains every instance of the set **`workerSkillPairs`** *that contains the list of skills of each worker.*

4. *Use the set* **`workerSkillPairs`** *to create an array* **`skillsWorkerList`**, *indexed over skills, that lists the workers capable of each skill.*

Here, you are doing the same thing, but this time listing each skill, then referencing all workers who have that associated skill.

Note that using a tuple structure for **`workerSkills`** *contributes to the sparsity of the model, because only the relevant worker/skill combinations are listed, as opposed to all possible combinations of workers and skills.*

In the data file:

1. *Populate* **`workerSkillPairs`** *from the database by using the following query:*

   ```
   select sa.name, s.skill_name from skill_availability
   sa, skills s, skill_group_skills sgs where
   sa.skill_group_id = sgs.skill_group_id and sgs.skill_id
   = s.skill_id;
   ```

Define decision variables and objective

1. *Define a Boolean decision variable* **`hireWorker[namesOfWorkers]`** *to indicate whether each worker is hired or not.*
2. *Define a float decision variable* **`workerSkillTime[workerSkillPairs]`** *to indicate how much time each worker spends on each skill.*
3. *Define the objective to minimize the number of workers hired.*

Define constraints

1. *Define a constraint* **`ctAvailability`** *that ensures that each individual worker's availability limit is met as follows:*

   ```
                forall  (w in namesOfWorkers)
                 ctAvailability : sum(s in
   workerSkillsList[w]) workerSkillTime[<w,s>] <=
   workerAvailability[<w>,<workerSkillGroup[w]>] *
   hireWorker[w];
   ```

2. *Define a constraint* **`ctMeetDemand`** *to ensure that the amount of time spent by all workers with a particular skill is at least as great as the demand for that skill.*

Post-processing for result output

1. *Create a list,* **`hiredWorkers`**, *of the names of the workers to be hired.*
2. *Note the use of an* **`execute`** *script block to write the list to the* ***Scripting log*** *output tab. .*
3. *Use* **`DBExecute`** *and* **`DBUpdate`** *statements to first clear the table of new hires and then populate it with the solution to your model.*

Solution

1. *Check the solution of both the* **`.mod`** *and the* **`.dat`** *files in the* **`<trainingDir>\OPL63.Labs\Staffing\Database_model\Solution`** *directory.*
2. *Run your model and make any necessary changes if it doesn't correspond to the solution.*
3. *Make a note of the solution.*

Summary

Review

In this lesson, you learned how OPL can use data stored in a spreadsheet or RDBMS:

- OPL can instantiate data by reading it from a spreadsheet or RDBMS.
- OPL can write the results of an optimization to a spreadsheet or RDBMS.
- OPL can use any legal SQL query to retrieve data from an RDBMS.

Lesson 9: Scheduling in OPL with CP Optimizer

OPL provides specialized keywords and syntax for modeling scheduling and allocation problems.

Instructor note
This lesson should last about 2 hours, including the practices.

This lesson introduces the concepts involved in describing a scheduling problem and the OPL keywords that facilitate modeling such a problem.

Introduction to scheduling

Learning objective
At the end of this lesson, you will be able to write a simple scheduling model in OPL. In this section, you gain an overview of how scheduling is implemented in OPL.

Key term
scheduling

OPL gives you access to IBM® ILOG® CP Optimizer features specially adapted to solving detailed scheduling problems over fine grained time. There are, for example, keywords particularly designed to represent such aspects as tasks and temporal constraints.

OPL offers you a workbench of modeling features in the IBM ILOG CP Optimizer engine that intuitively and naturally tackle the issues inherent in detailed scheduling problems from manufacturing, construction, driver scheduling, and more.

In a detailed scheduling problem, the most basic activity is assigning start and end times to intervals. IBM ILOG's implementation is especially useful for fine-grained scheduling.

Scheduling problems also require the management of minimal or maximal capacity constraints for resources over time, and of alternative modes to perform a task.

What is a detailed scheduling problem?

Detailed scheduling can be seen as the process of assigning start and end times to intervals, and deciding which alternative will be used if an activity can be performed in different modes. Scheduling problems also require the management of minimal or maximal capacity constraints for resources over time.

A typical scheduling problem is defined by:

- A set of time intervals -- definitions of activities, operations, or tasks to be completed, that might be optional or mandatory
- A set of temporal constraints – definitions of possible relationships between the start and end times of the intervals
- A set of specialized constraints – definitions of the complex relationships on a set of intervals due to the state and finite capacity of resources
- A cost function – for instance, the time required to perform a set of tasks, non execution cost of some optional tasks, or the penalty costs of delivering some tasks past a due date

Scheduling models in OPL

A scheduling model has the same format as other models in OPL:

- Data structure declarations
- Decision variable declarations
- Objective function
- Constraint declarations

OPL provides specialized variables, constraints and keywords designed for modeling scheduling problems.

A simple scheduling problem

Learning objective
Understand the scheduling framework as it is declared in OPL

Key terms
- interval decision variable
- intensity (calendar) function

The example that follows, a simple house building problem, declares a series of tasks (**`dvar interval`**) of fixed time duration (**`size`**) that need to be scheduled (assigned start and end times).

These tasks have **precedence constraints**. This means that one task must be completed before another can start. For example, in the example code, the **`carpentry`** task must be complete before the **`roofing`** task can start.

```
using CP;
dvar interval masonry size 35;
dvar interval carpentry size 15;
dvar interval plumbing size 40;
dvar interval ceiling size 15;
dvar interval roofing size 5;
dvar interval painting size 10;
dvar interval windows size 5;
dvar interval facade size 10;
dvar interval garden size 5;
dvar interval moving size 5;
```

```
subject to {
endBeforeStart(masonry, carpentry);
endBeforeStart(masonry, plumbing);
endBeforeStart(masonry, ceiling);
endBeforeStart(carpentry, roofing);
endBeforeStart(ceiling, painting);
endBeforeStart(roofing, windows);
endBeforeStart(roofing, facade);
endBeforeStart(plumbing, facade);
endBeforeStart(roofing, garden);
endBeforeStart(plumbing, garden);
endBeforeStart(windows, moving);
endBeforeStart(facade, moving);
endBeforeStart(garden, moving);
endBeforeStart(painting, moving);
}
```

In OPL, the unit of time represented by an interval decision variable is not defined. As a result, the size of the masonry task in this problem could be 35 hours or 35 weeks or 35 months.

Intervals – the tasks to schedule

In OPL, tasks, such as the activities involved in house building problem, are modeled as **intervals**, represented by the decision variable type **`interval`**. An interval has the following attributes:

- A start
- An end
- A size
- Can be optional
- An intensity (calendar function)

The time elapsed between the start and the end is the **length** of an interval.

The **size** of an interval is the time required to perform the task without interruptions.

An interval decision variable allows these attributes to vary in the model, subject to constraints.

Syntax:

```
dvar interval <taskName> <switches>
```

where **`<switches>`** represents one or more different modifying conditions to be applied to the interval.

Some of the switches available include:

- Setting a time window for the interval, for example:
  ```
  dvar interval masonry in 0..20;
  ```
- Providing different **length** to the interval from its **size**. A task may have a fixed size, but processing may be suspended during a break, so that the length is greater than the size.

  For example, the windows in the house example may take five days (size) to install, but if work stops over a two-day weekend, the length of the windows interval decision variable would be **`7`**. The declaration would then be:

  ```
  dvar interval windows size 5 in 0..7;
  ```
- Optionality: interval decision variables can be declared as optional. An optional interval may or may not be present in the solution.
  If landscaping were an unnecessary part of the house building, the interval decision variable **`garden`** would be declared as optional:

  ```
  dvar interval garden optional;
  ```

The declaration

```
dvar interval garden optional in 20..32 size 5;
```

declares that the task **`garden`**, if present, requires 5 time units to execute, and must start after time unit 20 and end before time unit 32.

Stated in everyday language, this declaration says that construction of a garden is not mandatory for building the house, but if it is to be done, (and assuming that the time units are days), the task requires five days to perform, and the five days of garden construction must happen between day 20 and day 32 in the house construction time line.

You will find the complete syntax for **`interval`** declarations in the *Language Quick Reference* manual of the documentation.

Functions on intervals

A number of functions are available with intervals. These are normally used in decision expressions (using the keyword, **`dexpr`**) to access an aspect of the interval. Some of these include:

- **`endOf`** – integer expression used to access the end time of an interval
- **`startOf`** – integer expression used to access the start time of an interval
- **`lengthOf`** – integer expression used to access the length of an interval
- **`sizeOf`** – integer expression used to access the size of an interval

- **`presenceOf`** – integer expression returning 1 if an optional interval is present, and 0 otherwise

All the functions related to scheduling can be found in the **OPL Functions** section of the *Language Quick Reference* manual of the documentation.

Intensity (calendar functions)

A calendar (or **intensity** function) can be associated with an interval decision variable. Intensity is a function that applies a measure of usage or utility over an interval length. For example, it can be used to specify the availability of a person or physical resource (such as a machine) during the interval.

Syntax:
`dvar interval <taskName> intensity F;`

where **`F`** is a stepwise function with integer values

- The intensity is 100% by default, and can not exceed this value (granularity of 100).
- If a task cannot be processed at all during a certain time window, such as a break or holiday, then the intensity for that time period is set to 0.
- When a task is processed part-time (this can be due to worker time off, interaction with other tasks, etc.) the intensity is expressed as a positive percentage.

Consider a task, for example, **`decoration`**, that is performed during an interval one week in length. In this interval a worker works five full days, one half day, and has one day off; the intensity function would be 100% for five days, 50% for one day, and zero for the last day.

You declare the intensity values using a linear stepwise function, via the OPL keyword **`stepFunction`**.

Interval size, length, and intensity are always related by the following:

size multiplied by granularity is equal to the integral of the intensity over the length of the interval.

Intensity can not exceed 100%, so interval size can never exceed the interval length.

Therefore, in the proceeding example, the interval length is seven days and size equals 5.5 work days, and would be declared as follows:

```
stepFunction F = stepwise(0—>1; 100->5; 50->6; 0->7);
dvar interval decoration size 5..5 in 1..7 intensity F;
```

Scheduling constraints

Learning objective
Understand how to use specialized constraints for scheduling

Key terms
- precedence constraint
- cumulative constraint
- cumulative function expression
- sequence decision variable
- no overlap constraint
- span constraint
- synchronize constraint

Precedence constraints

Precedence constraints are common scheduling constraints used to restrict the relative position of interval variables in a solution. These constraints are used to specify when one interval variable must start or end with respect to the start or end time of another interval. A delay, fixed or variable, can be included.

For example a precedence constraint can model the fact that an activity **a** must end before activity **b** starts (optionally with some minimum delay **z**).

List of precedence constraints in OPL:
- **endBeforeStart**
- **startBeforeEnd**
- **endAtStart**
- **endAtEnd**
- **startAtStart**
- **startAtEnd**

Example syntax:

```
startBeforeEnd (a,b[,z]);
```

Where the end of a given time interval **a** (modified by an optional time value **z**) is less than or equal to the start of a given time interval **b**:

```
s(a) + z ≤ s(b)
```

Thus, if the ceiling had to dry for two days before the painting could begin, you would write:

```
endBeforeStart(ceiling, painting, 2);
```

The meanings of these constraints are intuitive, and you can find complete syntax and explanations for all of them at **OPL, the modeling language > Constraints > Types of constraints > Constraints available in constraint programming** in the *Language Reference Manual*.

Cumulative constraints

In some cases, there may be a restriction on the number of intervals that can be processed at a given time, perhaps because there are limited resources available. Additionally, there may be some types of **reservoirs** in the problem description (cash flow or a tank that gets filled and emptied).

These types of constraints on resource usage over time can be modeled with constraints on **cumulative function** expressions. A cumulative function expression is a step function that can be incremented or decremented in relation to a fixed time or an interval. A cumulative function expression is represented by the OPL keyword **cumulFunction**.

Syntax:

```
cumulFunction <functionName> = <elementary_function_expression>;
```

where **<elementary_function_expression>** is a cumulative function expression that can legally modify a **cumulFunction**. These expressions include:

- **step**

- **`pulse`**
- **`stepAtStart`**
- **`stepAtEnd`**

A cumulative function expression can be constrained to model limited resource capacity by constraining that the function be less than the capacity:

```
workersUsage <= NbWorkers;
```

The value of a cumulative function expression is constrained to be nonnegative at all times.

Example of a pulse function

`pulse` – represents the contribution to the cumulative function of an individual interval variable or fixed interval of time. Pulse covers the usage of a cumulative or renewable resource when an activity increases the resource usage function at its start and decreases usage when it releases the resource at its end time.

```
cumulFunction f = pulse(u, v, h);
cumulFunction f = pulse(a, h);
cumulFunction f = pulse(a, hmin, hmax);
```

where the pulse function interval is represented by **`a`** or by the start point **`u`** and end point **`v`**. The height of the function is represented by **`h`**, or bounded by **`hmin`** and **`hmax`**

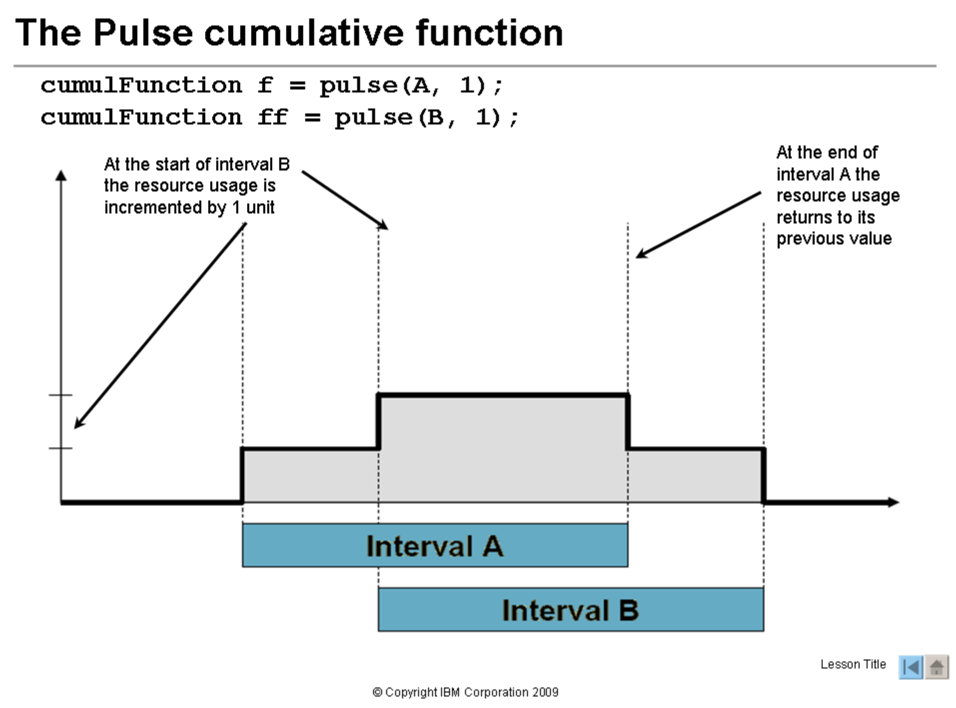
To illustrate, consider a cumulative resource usage function that measures how much of a resource is being used

- There are two intervals, A and B, bound in time
- Each interval increases the cumulative function expression by one unit over its duration

For each interval, this modification to the cumulative resource usage function can be made by incrementing the cumulative function with the elementary function, created with the interval and the given amount.

```
cumulFunction f = pulse(A, 1);
cumulFunction ff = pulse(B, 1);
```

Given this, the function would take the profile shown in the following graph:



The Pulse cumulative function

```
cumulFunction f = pulse(A, 1);
cumulFunction ff = pulse(B, 1);
```

At the start of interval B the resource usage is incremented by 1 unit

At the end of interval A the resource usage returns to its previous value

Example of step functions

step – represents the contribution to the cumulative function starting at a point in time:

```
cumulFunction f = step(u, h);
```

where the time **u** is the start of production or consumption and **h** represents the height of the function.

As another example, consider a function measuring a consumable resource, similar to a budget resource:

- The level of the resource is zero, until time 2 when the value is increased to 4. This is modeled by modifying the cumulative function with the elementary cumulative function **step** at time 2:

  ```
  cumulFunction f = step(2, 4);
  ```

- There are two intervals, A and B, fixed in time. Interval A decreases the level of the resource by 3 at the start of the interval, modeled by applying **stepAtStart**, created with Interval A and the value 3, to the cumulative function:

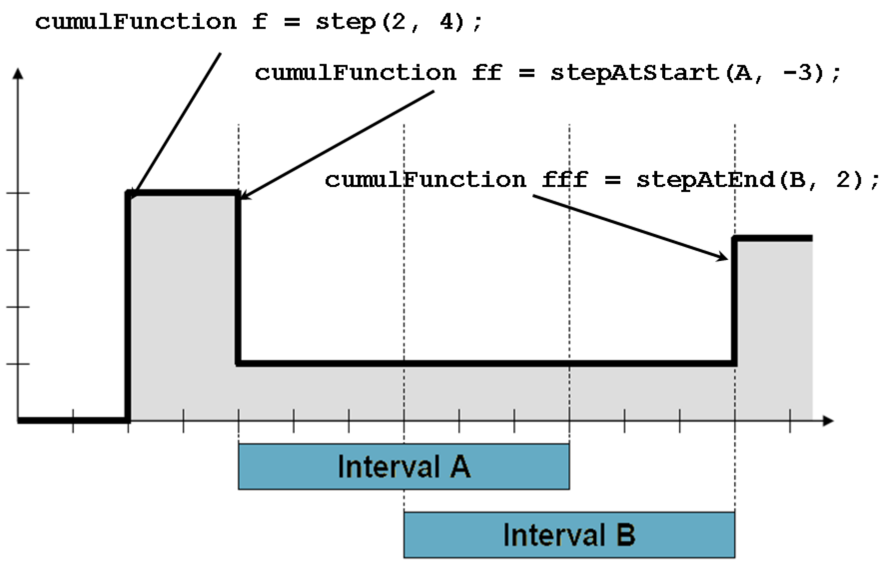
  ```
  cumulFunction ff = stepAtStart(A, -3);
  ```

- Interval B increases the level of the resource by 2 at the end of the interval, modeled by applying **stepAtEnd**, created with Interval B and the value 2, to the cumulative function for the interval:

  ```
  cumulFunction fff = stepAtEnd(B, 2);
  ```

Given this, the function would take the profile shown in the following graph:

Step cumulative functions


cumulFunction f = step(2, 4);
cumulFunction ff = stepAtStart(A, -3);
cumulFunction fff = stepAtEnd(B, 2);
Interval A
Interval B

Other cumulative function expressions

- **`stepAtStart`** – represents the contribution to the cumulative function beginning at the start of an interval:

  ```
  cumulFunction f = stepAtStart(a, h);
  cumulFunction f = stepAtStart(a, hmin, hmax);
  ```

  where the start of interval **`a`** is the start of production or consumption. The height of the function is represented by **`h`**, or bounded by **`hmin`** and **`hmax`**.

- **`stepAtEnd`** – represents the contribution to the cumulative function starting at the end of an interval:

  ```
  cumulFunction f = stepAtEnd(a, h);
  cumulFunction f = stepAtEnd(a, hmin, hmax);
  ```

  where the end of interval **`a`** is the start of production or consumption. The height of the function is represented by **`h`**, or bounded by **`hmin`** and **`hmax`**.

Sequence decision variable and no overlap constraints

A scheduling model can contain tasks that must not overlap, for example, tasks that are to be performed by a given worker cannot occur simultaneously.

To model this, you use two constructs:

- The **`sequence`** decision variable
- The **`noOverlap`** scheduling constraint

Unlike precedence constraints, there is no restriction on relative position of the tasks. In addition, there may be transition times between tasks.

The Sequence decision variable

Sequences are represented by the decision variable type **sequence**.

Syntax:

```
dvar sequence <sequenceName> in <intervalName> [types T];
```

where **T** represents a non-negative integer.

A sequence variable represents a total order over a set of interval variables. If a sequence **seq** is defined over a set of interval variables **{ a1, a2, a3, a4 }**, a value for this sequence at a solution can be: **(a1, a4, a2, a3).** A non-negative integer (the type) can be associated with each interval variable in the sequence. This integer is used by some constraints to group the set of intervals according to the type value.

Absent interval variables are not considered in the ordering.

Example:

```
dvar sequence workers[w in WorkerNames] in
all(h in Houses, t in TaskNames: Worker[t]==w) itvs[h][t] types
all(h in Houses, t in TaskNames: Worker[t]==w) h;
```

The sequence can contain a subset of the interval variables or be empty. In a solution, the sequence will represent a total order over all the intervals in the set that are present in the solution.

The assigned order of interval variables in the sequence does not necessarily determine their relative positions in time in the schedule. To control the relative positions in time of a sequence, you use the constraints:

- **before**
- **first**
- **last**
- **prev**
- **noOverlap**

Complete syntax and explanations for these constraints are at **OPL, the modeling language > Constraints > Types of constraints > Constraints available in constraint programming** in the *Language Reference Manual*.

You are now going to look at the **noOverlap** constraint.

No overlap constraints

To constrain the intervals in a sequence such that they:

- Are ordered in time corresponding to the order in the sequence
- Do not overlap
- Respect transition times

OPL provides the constraint **noOverlap**.

Syntax:

```
noOverlap (<sequenceName> [,M]);
```

where **<sequenceName>** is a previously declared sequence decision variable, and **M** is an optional transition matrix (in the form of a tuple set) that can be used to maintain a minimal distance between the end of one interval and the start of the next interval in the sequence.

In the following example:

- A set of **`n`** activities **`A[i]`** of integer type **`T[i]`** is to be sequenced on a machine.
- There is a sequence dependent setup time, **`abs(ti-tj)`** to switch from activity type **`ti`** to activity type **`tj`**.
- There should be no activity overlap.

```
{int} Types = { T[i] | i in 1..n };
tuple triplet { int id1; int id2; int value; };
{triplet} M = { <i,j,ftoi(abs(i-j))> | i in Types, j in Types };

dvar interval A[i in 1..n] size d[i];
dvar sequence p in A types T;

subject to {
  noOverlap(p, M);
};
```

An additional interesting use of **`noOverlap`** is to shortcut the creation of the interval sequence variable for simple cases where the sequence is not useful:

```
noOverlap(A);
```

is equivalent to:

```
dvar sequence p in A;
noOverlap(p);
```

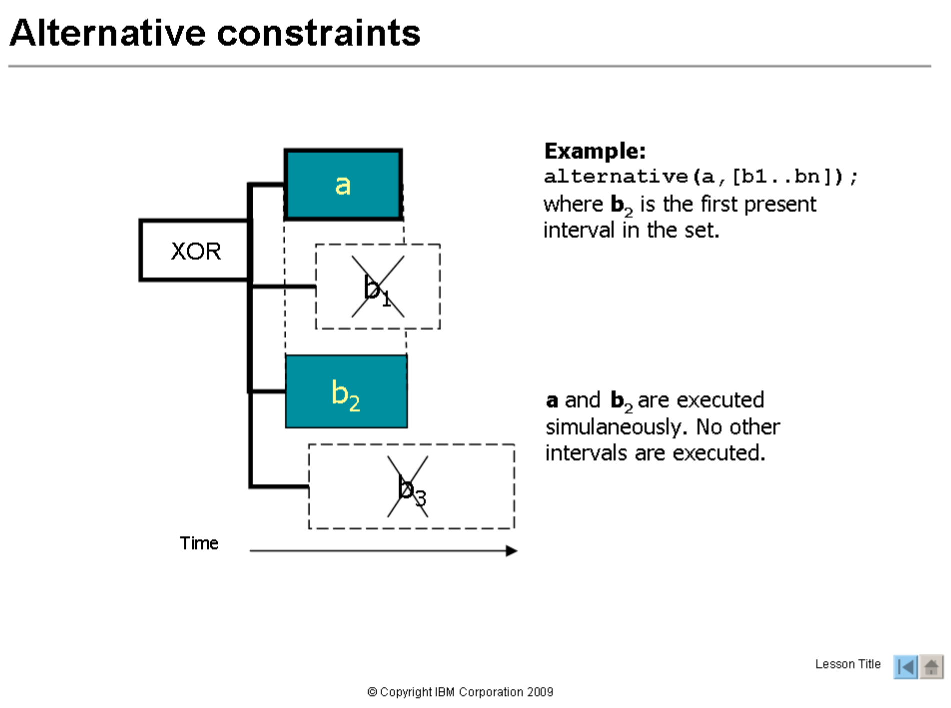
where **`A`** is an interval decision variable (or a set of intervals) and **`p`** is a sequence decision variable.

Alternative and span constraints

The two keywords **`alternative`** and **`span`** provide important ways to control the execution and synchronization of different tasks.

An **`alternative`** constraint between an interval decision variable **`a`** and a set of interval decision variables **`B`** states that interval **`a`** is executed if and only if exactly one of the members of **`B`** is executed. In that case, the two tasks are synchronized.

That is, interval **`a`** starts together with the first present interval from set **`B`** and ends together with it. No other members of set **`B`** are executed, and interval **`a`** is absent if and only if all intervals in the set **`B`** are absent, as shown in the following diagram:



Alternative constraints

Example:
`alternative(a,[b1..bn]);`
where **b**$_2$ is the first present interval in the set.

a and **b**$_2$ are executed simulaneously. No other intervals are executed.

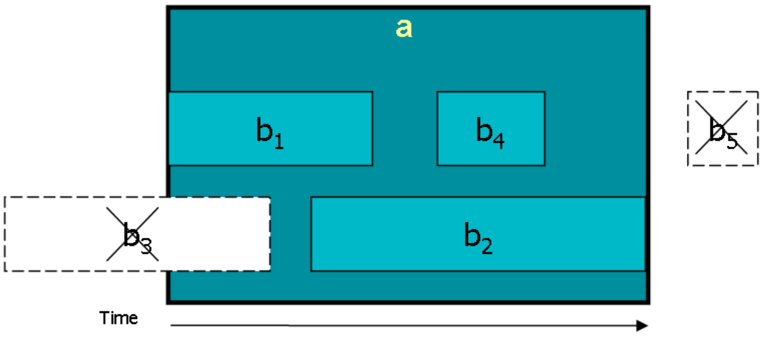
Lesson Title

© Copyright IBM Corporation 2009

A span constraint between an interval decision variable **a** and a set of interval decision variables **B** states that interval **a** spans over all intervals present in the set. That is: interval **a** starts together with the first present interval from set **B** and ends together with the last one. Interval **a** is absent if and only if all intervals in the set **B** are absent, as shown in the following diagram:

Span constraints

Example: `span(a,[b1..bn]);`



a starts together with **b_1** and ends together with **b_2**.
Intervals **b_3** and **b_5** are not executed.

Lesson Title

In both of these constraints, the array **`B`** must be a one-dimensional array; for greater complexity, use the keyword **`all`**.

Examples:

```
alternative(tasks[h] [t], all(s in Skills: s.task==t) wtasks[h]
[s]);

span(house[i], all(t in tasks : t.house == i) tasks[t]);
```

Synchronize constraint

A synchronization constraint (keyword **`synchronize`**) between an interval decision variable **`a`** and a set of interval decision variables **`B`** makes all present intervals in the set **`B`** start and end at the same times as interval **`a`**, if it is present.

The array **`B`** must be a one-dimensional array; for greater complexity, use the keyword **`all`**.

Example:

```
synchronize(task[i], all(o in opers : o.task == i) tiopers[o]);
```

Putting everything together - a staff scheduling problem

Learning objective
Use OPL scheduling keywords and syntax to model a simple staff scheduling problem

Key terms
- alternative resource
- resource pool
- surrogate constraint

The business problem

You are now going to use OPL to create a model representing a staff scheduling problem. This is a classic type of task-based scheduling problem to model.

A telephone company must schedule customer requests for installation of different types of telephone lines:

- First (or principal) line
- Second (or additional) line
- ISDN (digital) line

Each request has a requested due date; a due date can be missed, but the objective is to minimize the number of days late.

These three request types each have a list of tasks that must be completed in order to complete the request. There are precedence constraints associated with some of the tasks. Each task has a fixed duration and also may require certain fixed quantities of specific types of resources.

The resource types are

- Operator
- Technician
- CherryPicker (a type of crane)
- ISDNPacketMonitor
- ISDNTechnician.

The tasks types, along with their durations and resource requirements are outlined in the following table:

Task type	Duration	Resources
MakeAppointment	1	Operator
FlipSwitch	1	Technician
InteriorSiteCall	3	Technician x 2
ISDNInteriorSiteCall	3	• Technician • ISDNTechnician • ISDNPacketMonitor
ExteriorSiteCall	2	• Technician x 2 • CherryPicker
TestLine	1	Technician

Each request type has a set of task types that must be executed. Some of the tasks must be executed before other tasks can start:

Request type	Task type	Preceding tasks
FirstLineInstall	FlipSwitch	
	TestLine	FlipSwitch

Request type	Task type	Preceding tasks
SecondLineInstall	MakeAppointment	
	InteriorSiteCall	MakeAppointment
	TestLine	InteriorSiteCall
ISDNInstall	MakeAppointment	
	ISDNInteriorSiteCall ExteriorSiteCall	MakeAppointment
	TestLine	InteriorSiteCall, ExteriorSiteCall

The data

The following resources are available:

Resource	ID/Name
Operator	"Patrick"
Technician	"JohnTec"
Technician	"PierreT"
CherryPicker	"VIN43CP"
CherryPicker	"VIN44CP"
ISDNPacketMonitor	"EQ12ISD"
ISDNTechnician	"RogerTe"

The requests that need to be scheduled are:

Request Number	Request Type	Due date
0	FirstLineInstall	22
1	SecondLineInstall	22
2	FirstLineInstall	1
3	ISDNInstall	21
4	SecondLineInstall	21
5	ISDNInstall	22

Model the staff scheduling problem

You are now going to perform a series of steps in the **Staff Scheduling** workshop based on this problem.

What is the objective?

In business terms, we might say the objective is “to improve on-time performance.” However, this is a qualitative statement that is difficult to model. In order to find a modeling representation (i.e. quantifiable) of the idea, we need to turn things around; instead of maximizing something abstract, we find a number that needs to be kept to a minimum. Thus, our objective becomes:

To minimize the total number of late days (days beyond the due date when requests are actually finished).

What are the unknowns?

The unknowns are:

- When each task will start
- Which resource will be assigned to each task

Before you begin to work with the lab files, here is an overview of what you are going to do. Each task will be decomposed step by step to demonstrate the process.

How to model this situation:

1. The task type `FlipSwitch` requires a Technician, and there are two Technicians. For each task of this type, you create three `interval` decision variables. Two of these intervals are optional, meaning they may or may not appear in the solution.
2. To constrain such that when an optional interval is present, it is scheduled at exactly the same time as the task interval, use a `synchronize` constraint.
3. To ensure that the appropriate number of worker intervals are used, write a constraint that requires that the sum of that the sum of present worker intervals is equal to the number of resources required.

Details of the intervals for FlipSwitch

- One optional represents `JohnTec` being assigned to the task
- The other optional represents `PierreT` being assigned to the task.
- The third interval represents the task itself, and is used in other constraints.

In general, if there are multiple resources with identical properties, it is best to model them as a **resource pool**. However, one can imagine that in this example new constraints related to workers' days-off could be added, so here each worker is treated as an individual resource.

Resource pools are explained in the section on **surrogate constraints**.

Modeling the precedence constraints

To model that some tasks in a request must occur before other tasks in the same request, you use the precedence constraint `endBeforeStart`.

While the data for this problem does not require there to be any delay between tasks, you can add a delay to the model to allow for the possibility of a delay.

In the **Staff Scheduling** workshop, perform the **Declare task interval and precedences** step.

Now you will begin the hands-on part of this exercise, and build the staff scheduling model step by step.

You can perform this lab using the HTML workshop, or by following the instructions in the workbook. The HTML workshop will give you direct access to OPL documentation pages that can help you with the lab.

Declare task interval and precedences

Objective

- *Start building a scheduling model using some basic CP Optimizer constructs.*

Actions

- *Examine data and model files*
- *Define the tasks and precedences*
- *Solve the model and examine the output*

References

interval
endBeforeStart

Examine data and model files

1. *Import the **`sched_staffWork`** project into the OPL Projects navigator (Leave the **Copy projects into workspace** box unchecked) and open the **`step1.mod`** and **`data.dat`** files.*

   *The **`.mod`** file represents a part of what the finished model will look like. Most of the model is already done.*

   *Note that there is no objective function. At this point, you have what is called a **satisfiability problem**. Running it will determine values that satisfy the constraints, without the solution necessarily being optimal.*

2. *Examine closely how the data declarations for the model are formulated.*
3. *Note especially, the declaration of the set **`demands`**. This creates a set whose members come from the tuple **`Demand`**, which is a tuple of tuples (**`RequestDat`** and **`TaskDat`**). This set is made sparse by filtering it such that only task/request pairs that are found in the tuple set **`recipes`** are included. Effectively, it creates a sparse set of required tasks to be performed in a request and operations (the same tasks associated with a given resource). Only valid combinations are in the set.*

   This is a good example of the power of tuple sets to create sparse sets of complex data.

4. *The file **`sched_staff.dat`** instantiates the data as outlined in the problem definition. it instantiates*
   - ***`ResourceTypes`***
   - ***`RequestTypes`***
   - ***`TaskTypes`***
   - ***`resources`***
   - ***`requests`***
   - ***`tasks`***
   - ***`recipes`***
   - ***`dependencies`***
   - ***`requirements`***

   Discuss how model and data files are related with your instructor and fellow students.

Define the tasks and precedences

You are now ready to start declaring decision variables and constraints. At this point, we will define only the tasks, and the precedence rules that control them.

1. *Declare an interval decision value to represent the time required to do each request/operation pair in the set* **`demands`**. *Name the decision variable* **`titasks`**:

   ```
   dvar interval titasks[d in demands] size d.task.ptime;
   ```

2. *An important aspect of the modeling is expressing the precedence constraints on the tasks (demands). These constraints can be expressed using the constraint* **`endBeforeStart`**.

   The **`step1.mod`** *file already contains the preparatory declarations:*

   ```
   forall(d1, d2 in demands, dep in dependencies :
     d1.request == d2.request &&
     dep.taskb == d1.task.type &&
     dep.taska == d2.task.type)
   ```

   Examine these declarations with your instructor to understand clearly what they mean.

3. *Write the* **`endBeforeStart`** *constraint.*
4. *Compare with the solution in* **`<trainingDir>\OPL63.labs\Scheduling\Staff\solution\sched_staffSolution\step1.mod`**

Solve the model and examine the output

1. *Solve the model by right clicking the* **Step1** *run configuration and selecting* **Run this** *from the context menu.*
2. *Look at the results in the* **Solutions** *output tab. Can you determine what the displayed values represent?*
3. *Look at the* **Engine log** *and* **Statistics** *output tabs, and note that this model is, for the moment, noted as a "Satisfiability problem."*
4. *Close* **`Step1.mod`**.

Modeling the objective

To model the objective you need to determine the end time of each request. The request itself can be seen as an interval with a variable length. The request interval must cover, or span, all the intervals associated with the tasks that comprise the request.

You create the objective by finding the difference between the end time and the due date and minimizing it.

Now perform the next step of the **Staff Scheduling** workshop, **Compute the end of a task and define the objective** and the first substep of the **Define the resource constraints** step: **Review the needs**.

Compute the end of a task and define the objective

Actions

- *Compute the time needed for each request*
- *Transform the business objective into the objective function*
- *Solve the model and examine the output*

References

span
all
maxl

Compute the time needed for each request

1. *Open* **`Step2.mod`** *for editing.*
2. *The requests are modeled as interval decision variables. Write the following declaration in the model:*

   ```
   dvar interval tirequests[requests];
   ```

3. *Write a* **`span`** *constraint to link this decision variable to the appropriate* **`titasks`** *instances .*

   Use the **`all`** *quantifier to associate the required tasks for each request with the appropriate duration.*

4. *Check your solution against*
   `<trainingDir>\OPL63.labs\Scheduling\Staff\solution\sched_staffSolution\Step2.mod`

Transform the business objective into the objective function

The business objective requires the model to minimize the total number of late days (days beyond the due date when requests are actually finished). To do this in the model, you need to write an objective function that minimizes the time for each request that exceeds the due date.

1. *Calculate the number of late days for each request:*
   - *The data element* **`requests`** *is the set of data that instantiates the tuple* **`RequestDat`**. *The* **`duedate`** *is included in this information.*
   - *The interval* **`tirequests`** *represents the time needed to perform each request.*
   - *Subtract the* **`duedate`** *from the date on which* **`tirequests`** *ends.*

   Use the **`endof`** *function to determine the end time of* **`tirequests`**.

2. *Include a test that discards any negative results (requests that finish early) from the objective function.*

   Use the **`maxl`** *function to select the greater of:*
   - *the difference between due date and finish date*
   - *0*

3. *Minimize the sum of all the non-negative subtractions, as calculated for each request.*

Check the solution in
`<trainingDir>\OPL63.labs\Scheduling\Staff\solution\sched_staffSolution\Step2.mod`

Solve the model and examine the output

1. *Solve the model by right clicking the* ***Step2*** *run configuration and selecting* ***Run this*** *from the context menu.*
2. *Look at the* ***Engine log*** *and* ***Statistics*** *output tabs, and note that the model is now reported as a "Minimization problem," after the addition of the objective function. Scroll down a little further and notice the number of fails reported.*
3. *Look at the results in the* ***Solutions*** *output tab. You will notice that an objective is now reported, in addition to the values of* **`titasks`** *and* **`tirequests`**.

Review the needs

So far, you have defined the following constraints as identified in the business problem, as outlined in the workbook:

- *For each demand task, there are exactly the required number of task-resource* **`operation`** *intervals present.*

- *Each task precedence is enforced.*
- *Each* **`request`** *interval decision variable spans the associated* **`demand`** *interval decision variables.*

You now need to meet the following needs, not yet dealt with in the model:

- *Each task-resource* **`operation`** *interval that is present is synchronized with the associated task's* **`demand`** *interval.*
- *There is no overlap in time amongst the present* **`operation`** *intervals associated with a given resource.*
- *At any given time, the number of overlapping task-resource* **`operation`** *intervals for a specific resource type do not exceed the number of available resources for that type.*

Modeling the alternative resources

The task type **`FlipSwitch`** requires a technician. There are two technicians, i.e. there are two **alternative resources**, available to do the same task. You want to be able to optimize how each of these is used relative to the objective.

To create the alternative resources, you declare optional intervals for each possible task/resource pair.

To constrain a resource so that it cannot be used by more than one task at a given time, you need to ensure that there is no overlap in time amongst the intervals associated with a given resource.

How to model this situation:

1. Create a **`sequence`** decision variable from those intervals
2. Place a **`noOverlap`** constraint on the **`sequence`** decision variable.

Now perform the rest of the **Define the resource constraints** step.

Assign workers to tasks

It is now time to deal with the question of who does what. We know from the data that there is more than one resource, in some cases, capable of doing a given task. How do we decide who is the best one to send on a particular job?

The key idea in representing a scheduling problem with alternative resources is:

- *Model each possible task-resource combination with an optional interval decision variable.*
- *Link these with an interval decision variable that represents the entire task itself (using a* **`synchronize`** *constraint).*

1. *Open* **`Step3.mod`** *for editing.*
2. *You will see that a new data declaration has been added:*

   ```
   tuple Operation {
       Demand       dmd;
       ResourceDat  resource;
   };
   {Operation} opers = {<d, r >| d in demands, m in requirements, r
   in resources : d.task.type == m.task && r.type == m.resource};
   ```

   The members of the tuple set **`opers`** *are the set of tasks assigned to a resource.*

3. *There is also a new decision variable associated with this tuple set that calculates the time required for each* **`operation`***:*

   ```
   dvar interval tiopers[opers] optional;
   ```

Note that this variable is optional. If one of the optional interval variables is present in a solution, this indicates that the resource associated with it is assigned to the associated task.

Remember that in this model a task is called a **`demand`**, *and a task-resource pair is called an* **`operation`**.

4. *Declare a* **`sequence`** *decision variable named* **`workers`**, *associated with each resource.*

   Use **`all`** *to connect each* **`resource`** *used in an* **`operation`** *to its related* **`tiopers`** *duration:*

```
dvar sequence workers[r in resources] in all(o in opers :
o.resource == r) tiopers[o];
```

5. *Constrain this decision variable using a* **`noOverlap`** *constraint to indicate the order in which a resource performs its* **`operation`***s.*

"Just enough" constraint

Another constraint states that for each demand task, there are exactly the required number of task-resource operation intervals present ("just enough" to do the job – not more or less). The presence of an optional interval can be determined using the **`presenceOf`** *constraint:*

```
forall(d in demands, rc in requirements : rc.task == d.task.type) {
 sum (o in opers : o.dmd == d && o.resource.type == rc.resource)
presenceOf(tiopers[o]) == rc.quantity;
```

- *Write this into the model file.*

Check your work and solve the model

1. *Compare your results with the contents of* **`<trainingDir>\OPL63.labs\Scheduling\Staff\solution\sched_staffSolution\Step3.mod`**

   Do not yet copy the synchronization constraint into your **work** *copy. First you are going to solve the model and observe the results.*

2. *Solve the model by right clicking the* **Step3** *run configuration and selecting* **Run this** *from the context menu.*
3. *Look at the* **Engine log** *and* **Statistics** *output tabs, and note that the number of variables and constraints treated in the model has increased slightly.*
4. *The results in the* **Solutions** *output tab show values for four decision values now, as well as the solution.*

Synchronize simultaneous operations and observe the effects

1. *Declare a constraint that synchronizes each task-resource* **`operation`** *interval that is present with the associated task's* **`demand`** *interval:*

```
forall (r in requests, d in demands : d.request == r)
       synchronize(titasks[d], all(o in opers : o.dmd == d)
tiopers[o]);
```

2. *Solve the model by right clicking the* **Step3** *run configuration and selecting* **Run this** *from the context menu.*
3. *Look at the* **Engine log** *and* **Statistics** *output tabs. The number of variables and constraints treated in the model has increased significantly, as has the number of fails.*
4. *Look at the results in the* **Solutions** *output tab, and note, especially how values for* **`workers`** *have changed from the previous solve.*
5. *Close* **`Step3.mod`**.

Surrogate constraints

While these constraints completely describe the model, at times it is beneficial to introduce additional constraints that improve the search. Along with treating the resources individually, you can also treat the set of resources of a given type as a **resource pool**. A resource pool can be modeled using a cumulative function expression.

Each resource type has one **`cumulFunction`** associated with it. Between the start and end of a task, the **`cumulFunction`** for any required resource is increased by the number of instances of the resource that the task requires, using the **`pulse`** function.

A constraint that the **`cumulFunction`** never exceeds the number of resources of the given type is added to the model.

These surrogate constraints on the **`cumulFunction`** expressions are crucial as they enforce a stronger constraint when the whole set of resources of the tasks is not chosen.

Complete the **Staff Scheduling** workshop by performing the **Add a surrogate constraint to accelerate search** step.
You can perform this lab using the HTML workshop, or by following the instructions in the workbook. The HTML workshop will give you direct access to OPL documentation pages that can help you with the lab.

Add a surrogate constraint to accelerate search

Actions

- *Declare the cumulative function*
- *Constrain the cumulative function*
- *Solve the model and examine the results*

Reference

cumulFunction

Declare the cumulative function

1. *Open **`Step4.mod`** for editing.*
2. *To model the surrogate constraint on resource usage, a cumulative function expression is created for each resource type. Each **`cumulFunction`** is modified by a **`pulse`** function for each demand. The amount of the **`pulse`** changes the level of the **`cumulFunction`** by the number of resources of the given type required by the demand:*

```
cumulFunction cumuls[r in ResourceTypes] =
  sum (rc in requirements, d in demands : rc.resource == r &&
d.task.type == rc.task) pulse(titasks[d], rc.quantity);
```

Constrain the cumulative function

1. *You will see that a new intermediate data declaration exists:*

```
int levels[rt in ResourceTypes] = sum (r in resources : r.type ==
 rt) 1;
```

   This is used to test for the presence of a given resource in a resource type.

2. *Write a constraint that requires, when a resource is present in a resource type, that the value of the **`cumulFunction`** must not exceed the value of **`levels`**.*
3. *Compare your results with the contents of* ***`<trainingDir>\OPL63.labs\Scheduling\Staff\solution\sched_staffSolution\Step4.mod`***

Solve the model and examine the results

- *Solve the model by right clicking the **Step4** run configuration and selecting **Run this** from the context menu.*
- *If you look at the **Engine log** and **Statistics** output tabs, you will note a dramatic improvement in the number of fails, thanks to the surrogate constraint.*

A house building calendar problem

You are now going to consider a problem of scheduling the tasks involved in building multiple houses in such a manner that minimizes the overall completion date of the houses.

The business problem

There are five houses to be built. As usual, some tasks must take place before other tasks, and each task has a predefined size.

There are two workers, each of whom must perform a given subset of the necessary tasks.

A worker can be assigned to only one task at a time.

Each worker has a calendar detailing the days on which he does not work, such as weekends and holidays, with the following constraints:

- On a worker’s day off, he does no work on his tasks.
- A worker's tasks may not be scheduled to start or end on a day off.
- Tasks that are in process by the worker are suspended during his days off.

House Construction Tasks

Task	Size	Worker	Preceding tasks
Masonry	35	Joe	
Carpentry	15	Joe	masonry
Plumbing	40	Jim	masonry
Ceiling	15	Jim	masonry
Roofing	5	Joe	carpentry
Painting	10	Jim	ceiling
Windows	5	Jim	roofing
Facade	10	Joe	roofing plumbing
Garden	5	Joe	roofing plumbing
Moving	5	Jim	windows facade garden painting

What is the objective?

In business terms, the objective is to minimize the total number of days required to build five houses.

What are the unknowns?

The unknowns are when each task will start. The actual length of a task depends on its position in time and on the calendar of the associated worker.

What are the constraints?

The constraints specify that:

- A particular task may not begin until one or more given tasks have been completed
- A worker can be assigned to only one task at a time
- Tasks that are in process are suspended during the associated worker's days off
- A task cannot start or end during the associated worker's days off.

Modeling the workers' calendars

A **`stepFunction`** is used to model the availability (**`intensity`**) of a worker with respect to his days off. This function has a range of [0..100], where the value 0 represents that the worker is not available and the value 100 represents that the worker is available for a full work period with regard to his calendar.

While not part of this model, any value in 0..100 can be used as the intensity. For instance, the function could take the value 50 for a time window in which a resource works at half-capacity.

For each worker, a sorted tuple set is created. At each point in time where the worker's availability changes, a tuple is created. The tuple has two elements; the first element is an integer value that represents the worker's availability (0 for on a break, 100 for fully available to work, 50 for a half-day), and the other element represents the date at which the availability changes to this value. This tuple set, sorted by date, is then used to create a **`stepFunction`** to represent the worker's intensity over time. The value of the function after the final step is set to 100.

Go to the **House Building Calendar** workshop, and perform the step, **Define a calendar for each worker**.

Define a calendar for each worker

1. *Import the **`sched_calendarWork`** project into the OPL Projects Navigator (Leave the **Copy projects into workspace** box unchecked) and open the **`calendar.mod`** and **`calendar.dat`** files for editing.*
2. *Examine the first data declarations and their instantiations in the **`.dat`** file:*
    - *The first two declarations, **`NbHouses`** and **`range Houses`** establish simple declarations of how many houses to build, and a range that is constrained between 1 and that total number.*
    - *The next two declarations instantiate sets of strings that represent, respectively, the names of the workers and the names of the tasks to perform.*
    - *The declaration **`int Duration [t in TaskNames] = ...;`** instantiates an array named **`Duration`** indexed over each **`TaskNames`** instance.*
    - *The declaration **`string Worker [t in TaskNames] = ...;`** instantiates an array named **`Worker`** indexed over each **`TaskNames`** instance.*
    - *The tuple set **`Precedences`** instantiates task pairings in the tuple **`Precedence`**, where each tuple instance indicates the temporal relationship between two tasks: the task in **`before`** must be completed before the task in **`after`** can begin.*
    - *The tuple **`Break`** indicates the start date, **`s`**, and end date, **`e`** of a given break period. A list of breaks for each worker is instantiated as the array **`Breaks`**. Each instance of this array is included in a set named **`Break`**.*
3. *Declare a tuple named **`Step`** with two elements:*
    - *An integer value, **`v`**, that represents the worker's availability at a given moment (0 for on a break, 100 for fully available to work, 50 for a half-day)*
    - *An integer value, **`x`**, that represents the date at which the availability changes to this value. Make this element the key for the tuple.*

```
tuple Step {
int v;
key int x;
};
```

4. *Create a sorted tuple set such that at each point in time where the worker's availability changes, an instance of the tuple set is created. Sort the tuple set by date, and use a* **`stepfuncion`** *named* **`calendar`** *to create the intensity values to be assigned to each* **`WorkerName`**

   Use a **`stepwise`** *function*

   :

```
sorted {Step} Steps[w in WorkerNames] =
{ <100, b.s >| b in Breaks[w] } union
{ <0, b.e >| b in Breaks[w] };
stepFunction Calendar[w in WorkerNames] =
stepwise (s in Steps[w]) { s.v - >s.x; 100 };
```

   When two consecutive steps of the function have the same value, these steps are merged so that the function is always represented with the minimal number of steps.

Modeling the unknowns

You need to know the dates when each task will start. If you have that information, the end dates are known, since the size of the task is known and the breaks that will determine the length of the task are also known.

Write an expression that calculates the start date of each task for each house, using this information.

Associate the step function **`Calendar`** with an interval variable using the keyword **`intensity`** to take the worker's availability dates into account.

This has been done for you already in the workshop.

Continue with the **Declare the decision variable** step.

Declare the decision variable

1. *Continue looking at the model file – the following* **`interval`** *decision variable is declared:*

```
dvar interval itvs[h in Houses, t in TaskNames]
  size      Duration[t]
  intensity Calendar[Worker[t]];
```

2. *Can you see how the* **`Calendar`** *function is associated with the decision variable in order to ensure that the worker's availability is taken into account?*
3. *Discuss this with your instructor and fellow students.*

Modeling the objective

The objective of this problem is to minimize the total number of days required to build five houses. To model this, you minimize the overall completion date – i.e. the span of time from the start date of the first house to the completion date of the house that is completed last.

Return to the **House Building Calendar** workshop, and perform the step, **Define the objective function**.

Define the objective function

Action

- *Transform the business objective into the objective function*

Transform the business objective into the objective function

The business objective requires the model to minimize the total number of days required to build five houses. To do this in the model, you need to write an objective function that minimizes the maximum time needed to build each house and arrive at a minimum final completion date for the overall five-house project.

1. *Determine the maximum completion date for each individual house project using the expression **`endOf`** on the last task in building each house (the **`moving`** task) and*
2. *Minimize the maximum of these expressions.*

Check the solution in
`<trainingDir>\OPL63.labs\Scheduling\Calendar\solution\sched_calendarSolution\calendar.mod`

Modeling the precedence constraints

The precedence constraints in this problem are simple **`endBeforeStart`** constraints with no delay.

```
forall(p in Precedences)
      endBeforeStart(itvs[h][p.before], itvs[h][p.after]);
```

Practice this in the **Write the precedence constraint** substep of the step, **Define constraints** in the **House Building Calendar** workshop.

Write the precedence constraint

*The precedence constraints in this problem are simple **`endBeforeStart`** constraints with no delay.*

1. *Write a single constraint that can be applied via the tuple set **`Precedences`** to each instance of the interval decision variable **`itvs`**.*

   *Use filtering on **`(p in Precedences)`** to separate out start dates and end dates. Use arrays of the form **`[p.before]`** and **`[p.after]`**.*

2. *Check your work against the file*
   `<trainingDir>\OPL63.labs\Scheduling\Calendar\solution\sched_calendarSolution\calendar.mod`

Modeling the noOverlap constraint

To add the constraints that a worker can perform only one task at a time, the interval variables associated with that worker are constained to not overlap in the solution using the specialized constraint **`noOverlap`**:

```
forall(w in WorkerNames)
  noOverlap( all(h in Houses, t in TaskNames: Worker[t]==w) itvs[h][t]);
```

Go to the **House Building Calendar** workshop, and perform the substep, **Define constraints >Write the noOverlap constraint**.

Write the noOverlap constraint

1. *Write a constraint that says the interval variables associated with a worker are constrained to not overlap in the solution.*
2. *Check your work against the file*
   `<trainingDir>\OPL63.labs\Scheduling\Calendar\solution\sched_calendarSolution\calendar.mod`

*You may be surprised by the form of the **`noOverlap`** constraint in the solution. This form is a shortcut that avoids the need to explicitly define the interval sequence variable when no additional constraints are required on the sequence variable.*

Modeling forbidden start/end periods

When an intensity function is set on an interval variable, the tasks which overlap weekends and/or holidays will be automatically prolonged. An option could be available to start or end a task on a weekend day, but in this problem, a worker's tasks cannot start or end during the worker's days off.

A forbidden start or end is represented in IBM ILOG OPL by the constraints **`forbidStart`** and **`forbidEnd`**, which respectively constrain an interval variable to not end and not overlap where the associated step function has a zero value.

Go to the **House Building Calendar** workshop, and perform the substep, **Define constraints >Write the forbidden start/end period constraint** .

Write the forbidden start/end period constraint

1. *Write a constraint, using **`forbidStart`** and **`forbidEnd`**, that forbids a task to start or end on the associated worker's days off (i.e. when **`intensity`** = 0).*
2. *Check your work against the file* ***`<trainingDir>\OPL63.labs\Scheduling\Calendar\solution\sched_calendarSolution\calendar.mod`***

Matters of State: Understanding State Functions

Learning objective
At the end of this topic, you will have learned to use state functions and constraints in a CP Scheduling model

Key terms
- state function
- state constraint

In some cases, there may be a restriction on what types of tasks can be processed simultaneously. For instance, in the house building problem, a "clean task" like painting cannot occur at the same time as a "dirty" task like sanding the floors.

Moreover, some transition may be necessary between intervals with different states, such as needing to wait for the paint to dry before floor sanding can take place.

This type of situation is called a **state function**. A state function represents the changes in state over time, and can be used to define constraints.

OPL provides the keyword **`stateFunction`** to model this.

Syntax:

```
stateFunction <functionName> [with M];
```

where **`<functionName>`** is a label given to the function, and **`M`** is an optional transition matrix that needs to be defined as a set of integer triplets (just as for the **`noOverlap`** constraint). Thus this matrix is a tuple set.

For example, for an oven with three possible temperature levels identified by indexes 0, 1 and 2 we could have:

- **`[start=0, end=100): state=0`**
- **`[start=150, end=250): state=1`**
- **`[start=250, end=300): state=1`**
- **`[start=320, end=420): state=2`**
- **`[start=460, end=560): state=0,`**

In ordinary terms, this represents a set of non-overlapping intervals, each beginning at time **`start`** and ending at time **`end`**, over which the function maintains a particular non-negative integer value indicated by **`state`** (in this example, oven temperature).

In between those intervals, the state of the function is not defined (for example, between time 100 and time 150), typically because of an ongoing transition between two states (such as the time needed to change oven temperature from **`state 0`** to **`state 1`**).

To model the oven example:

```
tuple triplet {
 int start;
 int end;
 int state;
};
{ triplet } Transition = ...;
//...
stateFunction ovenTemperature with Transition;
```

State constraints

You can use constraints to restrict the evolution of a state function. These constraints can specify:

- That the state of the function must be defined and should remain equal to a given state everywhere over a given fixed or variable interval (**`alwaysEqual`**).
- That the state of the function must be defined and should remain constant (no matter its value) everywhere over a given fixed or variable interval (**`alwaysConstant`**).
- That intervals requiring the state of the function to be defined cannot overlap a given fixed or variable interval (**`alwaysNoState`**).
- That everywhere over a given fixed or variable interval, the state of the function, if defined, must remain within a given range of states **`[vmin, vmax]`** **`(alwaysIn)`**.

Additionally, **`alwaysEqual`** and **`alwaysConstant`** can be combined with synchronization constraints to specify that the given fixed or variable interval should have its start and/or end point synchronized with the start and/or end point of the interval of the state function that maintains the required state (notions of start and end alignment).

Example:

```
int MaxItemsInOven = ...;
int NbItems = ...;
range Items = 1..NbItems;
int DurationMin[Items] = ...;
int DurationMax[Items] = ...;
int Temperature[Items] = ...;
tuple triplet { int start; int end; int state; };
{ triplet } Transition = ...;

dvar interval treat[i in Items] size DurationMin[i]..DurationMax[i];

stateFunction ovenTemperature with Transition;
cumulFunction itemsInOven = sum(i in Items) pulse(treat[i], 1);

constraints {
itemsInOven <= MaxItemsInOven;
forall(i in Items)

alwaysEqual(ovenTemperature, treat[i], Temperature[i], 1, 1);
}

dvar interval maintenance ...;

constraints {
// ...
alwaysIn(ovenTemperature, maintenance, 0, 4);
}
```

This example models the oven problem described above (with more possible values for **`state`**), and adds the notion of synchronization. Certain items can be processed at the same time in the oven as they require the same temperature. There are limits, however, on the number of items that can be in the oven at one time, and on item size.

Finally, the last constraint models a required maintenance period where oven temperature cannot go beyond level 4.

A wood cutting problem

You are now going to perform a lab that uses state constraints. The problem involves a process: cutting different kinds of logs into wood chips.

The business problem

A wood factory machine cuts stands (processed portions of log) into chips. Each stand has these characteristics:

- length
- diameter
- species of wood

The following restrictions apply:

- The machine can cut a limited number of stands at a time with some restriction on the sum of the diameters that it can accept.
- The truck fleet can handle a limited number of stands at a given time.
- Stands processed simultaneously must all be of the same species.
- Each stand has a fixed delivery date and a processing status of one of:
  - standard
  - rush

  Any delay on a rush stand will cost a penalty.

The wood cutting company needs to minimize costs per unit time, and reduce penalty costs resulting from late deliveries of rush stands to a minimum.

What are the unknowns?

The unknowns are the completion date of the cutting of the stands. An interval variable is associated with each of the stands. The size of an interval variable is the product of the length of the stand and the time it takes to cut one unit of the stand's species.

What are the constraints?

The constraints are:

- At a given time, the machine can cut:
  - A limited number of stands
  - A limited sum of stand diameters
  - Only one species.
- The trucks can carry a limited number of stands.

What is the objective?

In business terms, the objective is to minimize the combined total of cutting costs (expressed as a function of time spent on the machine) and the cost for any penalties due to late delivery of rushed orders.

In mathematical terms, this objective translates to minimizing the sum of:

- the product of the maximum cutting time per stand and the cost per time unit
- the product of the length of rushed stands that are late and the cost per unit of length for being late

Go to the **Wood Cutting** workshop and perform the steps in order.

You can perform this lab using the HTML workshop, or by following the instructions in the workbook. The HTML workshop will give you direct access to OPL documentation pages that can help you with the lab.

Examine the completed parts of the model

Actions

- *Import the project*
- *Modeling the processing of the stands*
- *Modeling the quantity constraint*
- *Modeling the diameter constraint*
- *Modeling the fleet constraint*

Import the project

1. *Import the* ***<trainingDir>\OPL63.labs\Scheduling\Wood\work\sched_woodWork*** *project into the OPL Projects Navigator. Leave the* **Copy projects into workspace** *box unchecked.*
2. *Open the* **sched_wood.mod** *file for editing and examine it.*

Modeling the processing of the stands

An interval variable is associated with each of the stands. The size of an interval variable is the product of the length of the stand and the time it takes to cut one unit of the stand's species:

```
dvar interval a[s in stands] size (s.len * cutTime[s.species]);
```

Modeling the quantity constraint

The number of stands being processed at a time can be modeled by a cumulative expression function. Between the start and end of the interval representing the processing of the stand, the cumul function is increased by 1 using the **pulse** *function. A constraint that the cumul function never exceeds the stand capacity of the machine is added to the model:*

```
cumulFunction standsBeingProcessed = sum (s in stands) pulse(a[s], 1);

  standsBeingProcessed   <= maxStandsTogether;
```

Modeling the diameter constraint

The total diameter of the stands being processed at a time can be modeled by a cumulative function. Between the start and end of the interval representing the processing of the stand, the cumul function is increased by the diameter using the **pulse** *function. A constraint that the cumul function never exceeds the diameter capacity of the machine is added to the model:*

```
cumulFunction diameterBeingProcessed = sum (s in stands) pulse(a[s],
s.diameter);

  diameterBeingProcessed <= maxDiameter;
```

Modeling the fleet constraint

The constraint on the number of trucks being used can be placed on the cumul function for the number of stands being processed:

```
cumulFunction trucksBeingUsed = standsBeingProcessed;

  trucksBeingUsed        <= nbTrucks;
```

Define the one species constraint

Actions

- *Declare the state function*
- *Write an* **`alwaysEqual`** *constraint*

Reference

alwaysEqual

Declare the state function

In this model, the wood cutting company can profit from processing multiple stands at the same time in the same batch, provided that certain constraints are met. One of these is that the cutting machine can only process one species of wood at a time. To express this in the model, you are first going to declare a state function called **`species`**.

- *Do this now in the model file.*

Write an alwaysEqual constraint

1. *Write a constraint that says that the value* **`species`** *in each member of the tuple set* **`stands`** *is equal when being processed by the cutting machine.*

   Use the **`ord`** *keyword to order the species together, and the scheduling constraint* **`alwaysEqual`** *to constrain the state function* **`species`**.

2. *Check your work against the file*
   `<trainingDir>\OPL63.labs\Scheduling\Wood\solution\sched_woodSolution\sched_wood.mod`

Examine the objective function

Action

- *Transform the business objective into the objective function*

Transform the business objective into the objective function

The objective requires the model to minimize the sum of two calculations.

The first is the product of the maximum cutting time per stand and the cost per time unit.

- *In the model, the maximum cutting time per stand is defined by a decision expression using the* **`dexpr`** *OPL keyword:*

  ```
  dexpr int makespan =
    max (s in stands) endOf(a[s]);
  ```

- *The first part of the objective function calculates the product of* **`makespan`** *and the cost per time unit:*

  ```
  minimize makespan * (costPerDay / nbPeriodsPerDay)
  ```

The second quantity to be minimized is the product of the length of rushed stands that are late and the cost per unit of length for being late.

- *To calculate the length (in feet, in this case) of stands identified as “rush” orders that are to be delivered late, we use another decision expression, named* **`lateFeet`**:

  ```
  dexpr float lateFeet =
    sum (s in stands : s.rush == 1) s.len * (endOf(a[s])
  >s.dueDate);
  ```

- *We can now complete the objective function by adding the calculation of cost of late rushed footage. The entire objective function is:*

```
minimize makespan * (costPerDay / nbPeriodsPerDay)
  + costPerLateFoot * lateFeet;
```

Spend some time examining the solution in
<trainingDir>\OPL63.labs\Scheduling\Wood\solution\sched_woodSolution\sched_wood.mod

Pay special attention to how the state constraint interacts with the objective.

Summary

Review

In this lesson, you learned about scheduling problems and how they are modeled, including:

- Interval decision variables:
    - An interval variable has a start, an end, a size and a length, each of which may be fixed or variable.
    - An interval variable can be declared to be optional.
    - A calendar (step function) can be associated with an interval variable.
- Sequence decision variables:
    - A sequence variable takes an array of interval variables and fixes these in a sequence.
    - Such a sequence represents a total order over all the intervals in the set that are present in the solution.
- Specialized scheduling constraints:
    - No overlap constraints
    - Precedence constraints
    - Cumulative constraints
    - Calendar constraints
    - State constraints

You also learned how to work with the special tools provided by IBM ILOG CP Optimizer in OPL for modeling detailed scheduling problems.

Lesson 10: Integer and Mixed Integer Programming

Instructor note
This lesson should last about 1 hour, including the demonstration and practice.

A program that contains only integer decision variables is called an **Integer Programming (IP)** problem. In practice, problems that contain integer elements often also contain continuous elements. Such a problem is called a **Mixed Integer Programming (MIP)** problem.

OPL can address both these types of problems effectively.

This lesson introduces simple Mixed Integer Programming (MIP) techniques using OPL.

IP and MIP models in OPL

Learning objective
At the end of this lesson, you will be able to use OPL to model and solve a simple MIP.

Key terms
- Integer Programming (IP)
- Mixed Integer Programming (MIP)
- linear relaxation

Practice

The telephone production problem as an IP

Take another look at the telephone production problem. The way it was solved in your Lab, there could be a non-integer optimal solution.

Set up a new run configuration:

1. Open the project `..\OPL63.labs\Phones\work\phonesWork.`
2. Select the project in the OPL Projects Navigator. Then select the **File > Copy Files to Project** menu item from the menu bar to copy `..\OPL63.labs\phones\solution\phonesSolution\phones2.dat` to `..\OPL63.labs\Phones\work\phonesWork`
3. Create a new run configuration (it should be named `Configuration3`).

Populate the run configuration and solve the model:

1. Populate `Configuration3` with `phones1.mod` and `phones2.dat.`
2. Examine the data file. In this instance, available time on the painting and assembly machines is not an integer, nor is the profit per each type of phone.
3. Run the model using `Configuration3`. Leave the project open as you are going to return to it in the practice.

The model returns a non-integer solution:

```
production = [299.8 851.93];
```

In reality, you cannot produce 299.8 desk phones and 851.93 cell phones. Production values must be represented by integers.

For those who have taken the *Learning MP for OPL* training, you will have already taken a descriptive look at this problem.

To get an integer solution, perform the following steps:

1. Select the project `phonesWork` in the OPL Projects Navigator. Then Select the **File > Copy Files to Project** menu item from the menu bar to copy.
2. Create a new run configuration (it should be named `Configuration4`).
3. Populate `Configuration4` with the model `phones_MIP.mod` and the data file `phones2.dat.`
4. Examine the model. Note that the data element `Min` and the decision variable `production` have been changed from floating point to integer variables.
5. Run the model using `Configuration4.`

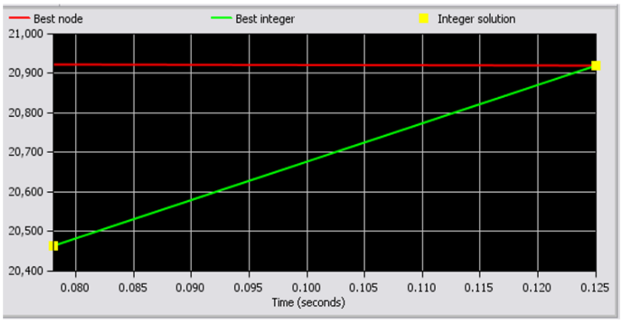
This time, an integer solution is returned: 299 desk phones and 852 cell phones. Examine these **Output** window tabs:

- **Solutions**
- **Engine Log**
- **Statistics**

In the **Engine Log** you can see that the MIP algorithm has been applied. What else do you learn from the information contained here?

The **Statistics** tab includes a progress chart showing how the algorithm progressively converged to the optimal integer solution:

Finding the integer solution



Lesson Title

Take a look at the other statistics. In a complex problem that requires a lot of time to run, what can you learn from looking at this tab?

Comparing solution time

OPL gives you tools for evaluating solution times.

Steps:

1. Return to the modified project, and create another run configuration, **`Configuration5.`**
2. Populate **`Configuration5`** with **`phones_MIP.mod`** and **`phones1.dat.`**
3. Run the model using **`Configuration5`** and examine the **Profiler** tab. Notice the total time.
4. Now run the model again, but use **`Configuration2.`**
5. Examine the **Profiler** tab. Notice the difference in time.

Using the data in **`phones1.dat`** produces an integer solution with both Simplex and MIP algorithms. But the Simplex algorithm is more efficient for solving this problem. While the time difference in this example is relatively insignificant, in very large problems, it can make a big difference. The declaration of **`int`** data will generally force the use of the MIP algorithm. In some cases it may be more efficient to declare **`float`** data, even if the values to be used are integers.

In a real-world situation, for products as small as telephones, with relatively low cost, it is probably more efficient to just round down from a fractional solution than to perform a MIP run. However, if you are manufacturing large, expensive items in quantity, for example, yachts, helicopters or construction cranes, you need an integer solution to avoid wasting resources, time and money.

Instructor note

Feel free to use this practice as an entry to a more complete discussion of how the students can use both statistics and profiler tabs to check the efficiency of their models. The tutorial suggested at the end of the final practice is a very good way to direct students toward more information on the progress chart without taking up class time, as it is quite complete, and is a follow-on to the warehouse problem presented in this lesson.

Linear relaxation

MIP algorithms make use of a linear relaxation of the model during the solution process. A linear relaxation of a MIP (or an IP), is an equivalent model, except that the integer requirement on the decision variable has been removed so that the model becomes an LP. A linear relaxation of a model can help:

- Prove the absence of a solution
- Search for the integer solution near a known fractional solution

You can use IBM® ILOG® Script flow control to create a linear relaxation of a model by using the **convertAllIntVars** method as shown below (flow control with IBM ILOG Script is discussed in detail in another optional lesson):

```
main {
   //converts IP/MIP to LP
    thisOplModel.convertAllIntVars();
    thisOplModel.generate();
    cplex.solve();

    writeln("Relaxed Model");
    writeln("OBJECTIVE: ",cplex.getObjValue());
}
```

Relaxation is only available to models that use the CPLEX® engine.

A warehouse allocation model

Learning objective
Gain a better understanding of IP through a practical example.

Key term
Combinatorial problem

A company is considering a number of locations for building warehouses to supply its existing stores. Each possible warehouse has a fixed maintenance cost and a maximum capacity specifying how many stores it can support. Each store can be supplied by only one warehouse and the supply cost to the store differs according to the warehouse selected. There are 5 warehouse locations, and 10 stores. The problem is to allocate warehouses to stores at minimum cost.

The fixed costs for the warehouses are all identical and equal to 30. The workshop contains a table that shows the 5 locations with their respective capacities and supply costs.

This type of problem can be modeled as an IP.

Practice

An IP solution to the warehouse problem

Perform the **Warehouse location** workshop

You can perform this lab using the HTML workshop, or by following the instructions in the workbook. The HTML workshop will give you direct access to OPL documentation pages that can help you with the lab.

Warehouse location

Problem Description

In this exercise you will model a warehouse location problem using integer programming and Boolean decision variables.

Principles of the problem:

- *A company is considering a number of locations for building warehouses to supply its existing stores.*
- *Each possible warehouse has a fixed cost associated with opening the warehouse, as well as a unique cost associated with assigning a store to a warehouse.*
- *Each warehouse has a maximum capacity specifying how many stores it can support.*
- *Each store can be supplied by only one warehouse.*
- *The decisions to be made are whether to open each warehouse or not, and which stores to assign to each open warehouse, while minimizing the total cost, which is the sum of the fixed opening costs and cost of assigning each store to each warehouse.*

Data:

- *There are 5 warehouses and 10 stores.*
- *The fixed costs for the warehouses are all identical and equal to 30.*
- *The following table shows the 5 locations with their respective capacities and costs of assigning each store to each warehouse.*

	Bonn	***Bordeaux***	***London***	***Paris***	***Rome***
capacity	*1*	*4*	*2*	*1*	*3*
store 1	*20*	*24*	*11*	*25*	*30*
store 2	*28*	*27*	*82*	*83*	*74*
store 3	*74*	*97*	*71*	*96*	*70*
store 4	*2*	*55*	*73*	*69*	*61*
store 5	*46*	*96*	*59*	*83*	*4*
store 6	*42*	*22*	*29*	*67*	*59*
store 7	*1*	*5*	*73*	*59*	*56*
store 8	*10*	*73*	*13*	*43*	*96*
store 9	*93*	*35*	*63*	*85*	*46*
store 10	*47*	*65*	*55*	*71*	*95*

Exercise folder

`<trainingDir>\OPL63.Labs\Warehouse\work`

Modeling the problem using IP

Objective

- *Use IP principles and Boolean decision variables to create a model optimizing warehouse allocation*

Action

- *Define constraints*

References

integer programming
mixed integer-linear programming
boolean expressions

Use Boolean decision variables

- *Import the* `warehouseWork` *project and examine the model file.*
- *The key idea in representing a warehouse-location problem as an integer program is to use a Boolean (1–0 or true/false) decision variable for each (warehouse, store) pair to represent whether a warehouse supplies a store:*
  ```
  dvar boolean Supply[Stores][Warehouses];
  ```
  In other words, `Supply[s][w]` *is* `1` *if warehouse* `w` *supplies store* `s` *and zero otherwise.*
- *In addition, the model also associates a decision variable with each warehouse to indicate whether the warehouse is open:*
  ```
  dvar boolean Open[Warehouses];
  ```

Define the objective function

The objective function

```
minimize
  sum(w in Warehouses) FixedCost * Open[w] +
  sum(w in Warehouses, s in Stores) SupplyCost[s][w] *
Supply[s][w];
```

expresses the goal that the model minimizes the fixed cost of the selected (i.e. open) warehouses and the supply costs of the stores.

Define constraints

The constraints state that:

- *Each store must be supplied by a warehouse*
- *Each store can be supplied only by an open warehouse*
- *No warehouse can deliver more stores than its allowed capacity*

The most delicate aspect of the modeling is expressing that a warehouse can supply a store only when it is open. This constraint can be expressed by inequalities of the form:

```
forall(w in Warehouses, s in Stores)
      Supply[s][w] <= Open[w];
```

This ensures that when warehouse `w` *is not open, it does not supply store* `s`*. This follows from the fact that* `open[w] == 0` *implies* `supply[w][s] == 0`.

As an alternative, you can write:

```
forall(w in Warehouses)
     sum(s in Stores) Supply[s][w] <= Open[w]*Capacity[w];
```

This formulation implies that a closed warehouse has no capacity.

- *Look at the model carefully and discuss with your instructor and fellow trainees how the model is constructed.*
- *Write the remaining constraints.*

Define instance data

- *The file* **`warehouse.dat`** *defines the instance data as shown in the table in the problem definition.*
- *It declares the warehouses and the stores, the fixed cost of the warehouses, and the supply cost of a store for each warehouse.*
- *Discuss how model and data files are constructed with your instructor and fellow trainees.*
- *Run the model and examine the results.*

Solution to the IP model

Solution with data declarations, decision variables, objective function and constraints

```
/*******
* Data *
*******/

{string} Warehouses = ...;
int NbStores = ...;
range Stores = 0..NbStores-1;

int FixedCost = ...;                            // fixed cost
for opening a warehouse
int Capacity[Warehouses] = ...;              // maximum number
 stores assigned to each warehouse
int SupplyCost[Stores][Warehouses] = ...;     // supply cost
between each store and each warehouse

/********************
* Decision variables *
********************/

dvar boolean Open[Warehouses];               // 1 if warehouse
 is open, 0 otherwise
dvar boolean Supply[Stores][Warehouses];      // 1 if store
supplied by warehouse, 0 otherwise

/********************
* Objective function *
********************/

minimize
  sum( w in Warehouses ) FixedCost * Open[w]
  + sum( w in Warehouses , s in Stores ) SupplyCost[s][w] *
Supply[s][w];
```

```
/**************
* Constraints *
**************/

subject to{

  forall( s in Stores )
    ctEachStoreHasOneWarehouse: sum( w in  Warehouses )
Supply[s][w] == 1;

  forall( w in Warehouses, s in Stores )
    ctUseOpenWarehouses: Supply[s][w] <= Open[w];

  forall( w in Warehouses )
    ctMaxUseOfWarehouse: sum( s in Stores ) Supply[s][w] <=
Capacity[w];
}

// this script is only for clearer output in the Console and
 serves no computational purpose
{int} storesof[w in Warehouses] = { s | s in Stores :
Supply[s][w] == 1 };
execute {
   writeln("Open=",Open);
   writeln("storesof=",storesof);
}
```

Summary

Review

In this lesson, you learned:

- Integer Programming (IP) can be used to address mathematical programming problems that include only integer decision variables.
- Mixed Integer Programming (MIP) is most used to address mathematical programming problems that include both integer and continuous decision variables. Many real-world problems are MIPs
- While Linear Programming (LP) can be used when results can be fractional, IP or MIP is often required when all or part of a result has to be expressed as an integer, for example the number of discrete units of a product to manufacture.
- In general, IP and MIP problems require more computational power to solve than pure LP's.
- A linear relaxation of an IP or a MIP can be used to try and construct an integer solution if no integer solution can be found

You have also had hands-on experience in modeling and solving IP problems using OPL.

Next steps

For a tutorial that will explore further MIP developments of this example and provide a real-time example of how to use the progress chart, refer to the online documentation: **IDE Tutorials > Tutorials > Tutorial: Statistics and Progress Chart > The Scalable Warehouse Example.**

Lesson 11: Piecewise Linear Problems

In business, problems don't always conform to simple, regular models. Customer demand changes with the seasons (or business cycles); availability of components can vary based on suppliers' stock; prices are subject to all sorts of economic factors and currency fluctuations.

These fluctuations can be nonlinear and often discontinuous. You can take them into account using models with **piecewise linear** functions.

OPL provides piecewise functions to handle problems with these characteristics.

Instructor note
This lesson should last about 1 hour.

Modeling piecewise linear functions

Learning objective
Learn how piecewise linear functions can be modeled in OPL

Key terms
- piecewise linear function
- breakpoint
- slope

What is a piecewise linear function?

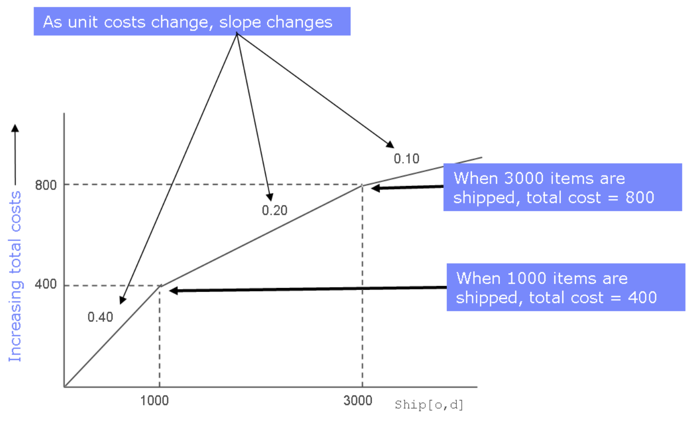
A piecewise linear function is defined as a nonlinear function that can be represented by a sequence of discrete segments, each of which is a linear function.

A piecewise transportation problem

If you took the *Learning MP for OPL* training course, you will recall the transportation problem in which the transportation cost between two locations **`o`** and **`d`** depends on the size of the shipment, **`ship[o][d]`**:

- The first 1000 items have a shipping cost of 0.40 each
- For the next 2000 items (i.e. items 1001-3000) the cost is 0.20 per item
- From item 3001 on, the cost decreases to 0.10 per item.
  The cost of each quantity bracket remains intact (i.e. the cost per unit changes only for additional units, and remains unchanged for the previous quantity bracket). Therefore, within each bracket there is a linear relationship between cost and quantity, but at each **breakpoint** the rate of linear variation changes, as shown in this diagram:

Unit costs vary with quantity



Lesson Title

© Copyright IBM Corporation 2009

The diagram shows that the total shipping cost is evaluated by 3 different linear functions, each determined by the quantity shipped:

- **`0.40 * items`** when **`ship[o][d]`** <= 1000
- **`0.40 * 1000 + 0.20 * (ship[o][d] - 1000)`** when **`ship[o][d]`** <=3000
- **`0.40 * 1000 + 0.20 * 2000 + 0.10 * (ship[o][d] - 3000)`** otherwise

This is an example of a typical piecewise linear function.

Modeling the piecewise transportation problem in OPL

OPL provides the keyword **`piecewise`** to help program this type of model. A piecewise linear function can be specified by giving:

- A set of **slopes** which represent the linear variation for each linear piece
- A set of **breakpoints** at which the slopes change
- The value of the functions at a given point

The syntax for piecewise linear functions is:

```
piecewise{s1 -> b1;s2 -> b2;...sn}(<knownValue>,<valuePoint>)
<value>;
```

where:

- **`s1`** is a slope that applies up to breakpoint **`b1`**
- **`s2`** is a slope that applies up to breakpoint **`b2`**
- **`sn`** is the last slope, which applies to all cases not covered by slope/breakpoint pairs **`1..(n-1)`**
- **`<knownValue>`** is the value of the data element **`<value>`** that is known at point **`<valuePoint>`**
- **`<value>`** is the data element that provides the breakpoints for the piecewise linear function.

Practice

The transportation function is a piecewise linear function of **`ship[o][d]`**.

Can you determine the values of the these aspects of the piecewise function?

- Slopes
- Breakpoints
- The value at point 0

How would you declare this function in an OPL model?

Instructor note
The values are:

- Slopes 0.4, 0.2, and 0.1
- Breakpoints 1000 and 3000
- The value 0 at point 0

It can thus be specified in a model file as:

```
piecewise{0.4 -> 1000;0.2 -> 3000;0.1}(0,0) ship[o][d];
```

By default, OPL assumes that a piecewise linear function evaluates to zero at the origin, so that the piecewise linear function could actually be written omitting the **`(0,0)`**.

This example has a fixed number of pieces, but OPL also allows the number of pieces to be generically instantiated. The number of pieces may then depend on the input data, as in

```
piecewise(i in 1..n) {
   slope[i] -> breakpoint[i];
   slope[n+1];
} ship[o][d];
```

There may be several different generic pieces in a piecewise linear function. It is important to stress that:

- Breakpoints and slopes in piecewise linear functions must not contain decision variables, since these variables have no value at the current stage of the computation.
- The breakpoints must be strictly increasing.

Example of a piecewise linear function in the objective

The piecewise linear expression:

```
maximize piecewise(i in 1..n)
{slope[i] -> breakpoint[i]; slope[n+1]}(0,objectiveforxequals0) x;
```

describes a piecewise linear function of x. The function has slopes 1, 2, and -3, breakpoints 100 and 200, and evaluates to 300 at point 0

Discontinuous piecewise linear functions

A piecewise linear function need not be continuous. A **discontinuous piecewise linear function** occurs when, in the syntax of a piecewise linear function with slopes and break points, two successive breakpoints are identical and the value associated with the second one is considered to be a "step" instead of a continuous "slope." OPL allows you to write discontinuous as well as continuous piecewise linear functions.

The syntax is as follows:

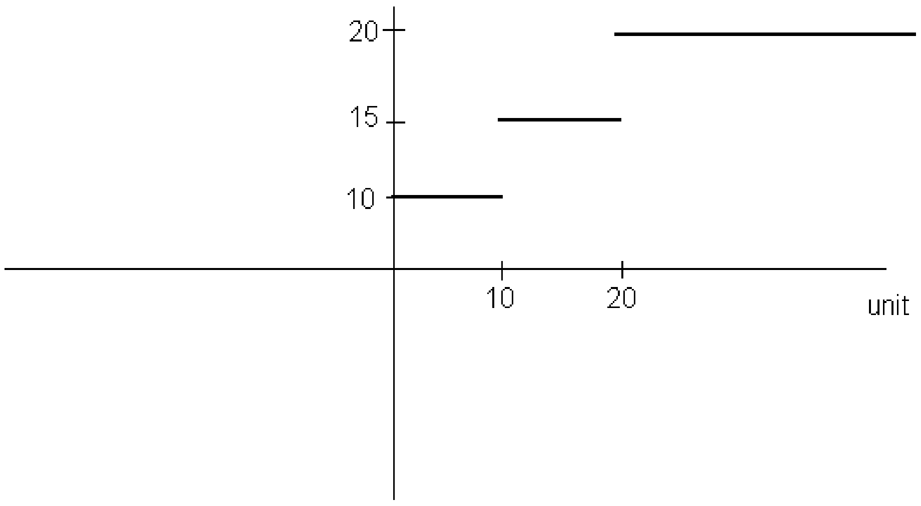
```
piecewise{s1 -> b1;step1 -> b1;s2 -> b2;step2 ->
b2;...sn}(<knownValue>,<valuePoint>) <value>;
```

where all entities are defined as for continuous piecewise, and **`step1`** is the difference between the value of **`s2`** at breakpoint **`b1`** and **`s1`** at **`b1, and so on.`**

For example, assume that the cost values for each possible value of **`unit`** are as shown in the following table:

Values of `unit`	Cost
<0	0
0 to 10	10
10 to 20	15
>20	20

If you graph this out, you get the following plot:



Discontinuities occur at the breakpoints of 10 and 20. In each case, each discontinuity has a **`step`** of 5. The following OPL declaration represents the discontinuous cost shown in the example above:

```
piecewise{0->10; 5->10; 0->20; 5->20; 0} (0,10) unit;
```

Practice

Go to the **Pasta Production and Delivery** lab and complete the step, **Factor in a piecewise cost change**.

You can perform this lab using the HTML workshop, or by following the instructions in the workbook. The HTML workshop will give you direct access to OPL documentation pages that can help you with the lab.

Factor in a piecewise cost change

Objective

- *Redesign the model to take a discontinuous cost change into account.*

Action

- *Model the cost break*

References

piecewise linear programming
discontinuous piecewise linear programming (scroll down)

Problem description

*Now, imagine that your pasta production facility is getting old, and when you make more than 20 units in a batch, the machine overheats and causes the production cost to go up by a factor of three! This not only changes the inside production cost, it introduces a **breakpoint** where the price takes a sudden, **piecewise** jump.*

The problem, then, is to change the model to take the breakpoint and cost change into account.

Model the cost break

Perform the following steps:

1. *Import the **`<trainingDir>\OPL63.labs\Pasta\PWL\work\productWork`**. project into the OPL Projects Navigator*
2. *Since the overall cost of inside production now has to be calculated differently than the overall cost of outside production, it is necessary to define them separately. Write two reusable expressions that use the decision variables **`insideProduction`** and **`outsideProduction`** to calculate these values. Name the expressions **`overallInsideCost`** and **`overallOutsideCost`** respectively. **Hint:**use the **`dexpr`** keyword for each of the expressions, and calculate **`overallInsideCost`** using the **`piecewise`** keyword.*
3. *Rewrite the objective function to minimize the sum of the two overall costs.*
4. *Run the model and see how the result differs from your original model.*
5. *Compare your model to the one found in **`<trainingDir>\OPL63.labs\Pasta\PWL\solution\productSolution`**.*

Summary

Review

In this lesson, you learned about:

Piecewise linear functions:

- A piecewise linear function is defined as a nonlinear function that can be represented by a sequence of discrete segments, each of which is a linear function. It can be specified by:
    - A set of slopes which represent the linear variation for each linear piece
    - A set of breakpoints at which the slopes change
    - The value of the functions at a given point
- A piecewise linear function can be discontinuous.

You also learned how to work with the special tools provided in OPL for modeling these types of problems.

Lesson 12: Network Models

Your pasta has been produced, now it has to be delivered. Your example problem, like real business problems, involves more than one set of requirements, and involves different kinds of data and structures.

Instructor note
This lesson should last 1 hour 30 minutes, including the practices.

Delivering a product over a **network** of roads requires specialized modeling structures. OPL can model a network, and, when it is advantageous, can use a specialized network version of the simplex algorithm to solve it.

Product delivery: a network problem

Learning objective
Discover the particularities of a network model and how OPL deals with this type of structure

Key terms
- network structure
- arc
- node

The business problem

The pasta company has internal production sites, and an external production partner with its own site. Products produced in these diverse locations must be delivered to warehouses of the distributor for onward delivery to customers.

- A road network links the production sites and the warehouses (**nodes**).
- The company has established routes it uses (**arcs**), with known costs associated.
- The company wants to know how many of each product to ship (i.e. how much to ship from each site) over each existing arc, in order to meet demand and minimize costs.

What are the unknowns?

- How many of each product to ship:
    - From each manufacturing site
    - To each warehouse
- Which arc to use for each order?

- Minimize cost
- Meet customer demand

Modeling network structures

What makes this problem different from those you have seen until now is the notion of a **node** (production location or warehouse) connected by an **arc** (link). The nodes are fixed, but many different arcs are possible to connect them. This kind of problem is said to have a **network structure**, and OPL can use LP to solve such problems. When the user specifies it, OPL can use specialized algorithms to solve network problems.

What is the network structure of this problem?

This problem involves planning product delivery from manufacturer to distributor in such a way as to satisfy the objectives. The assumption remains that all produced product will be delivered, and that production exactly equals demand.

This network structure is composed of 2 layers:

- Production - includes 2 inside and 1 outside production locations. Each production location can manufacture and distribute all of the products.
- Distribution - 7 warehouses

The elements to be manipulated are:

- Internal production locations - 2 nodes belonging to the pasta company
- External production locations - 1 node belonging to the outside suppliers
- Warehouses - fixed locations where goods are sent for distribution to customers.
- Arcs - each arc is a trip for a truck between a production location and a warehouse, i.e. links
- Cost per arc

What are the decision variables?

Decision variable	Representation in OPL
Total amount of product shipped from internal production sites to warehouses.	`dvar float insideFlow[Products][Locations][Warehouses]`
Total amount of product shipped from external production site to warehouses.	`dvar float outsideFlow[Products][Warehouses]`

What are the data elements?

Data element	Representation in OPL
Production locations	`{string} Locations`
Distributor warehouses	`{string} Warehouses`
Table of arcs and costs from internal sites to warehouses	`int internalArc[Locations][Warehouses]`
Table of arcs and costs from external sites to warehouses	`int externalArc [Warehouses]`

What are the objectives?

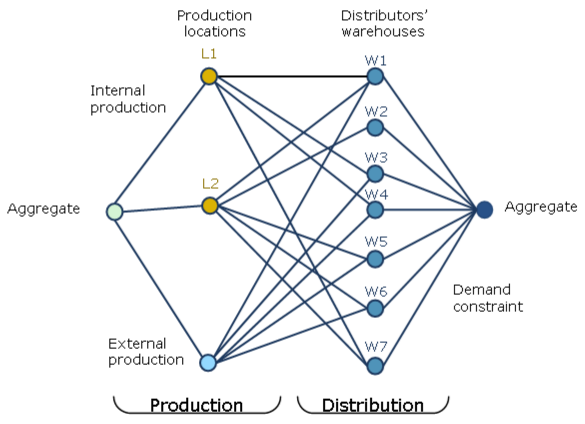
To minimize the shipping costs and know how much of each product to ship:

- from internal sites to **`Warehouses`** using arcs **`internalProduction`**
- from the external site to **`Warehouses`** using arcs **`externalProduction`**

Note that because there is only one external production site, we do not actually need to define the origin node in the external arcs.

An instance of the problem can be represented as a diagram:

Network flow problem



In this diagram:

An instance of the entity...	Is represented by...
`Locations`	A node on the **Production** side, for example, **L2**
`Warehouses`	A node on the **Distribution** side, for example, **W2**
`internalArc`	A path between an **Internal Production** site and a **Warehouse,** for example, **L1 - W4**
`externalArc`	A path between an **External Production** site and a **Warehouse,** for example, **W7**

What are the constraints?

In order to represent product deliveries to the warehouses, we need to add some constraints to:

- specify that deliveries can only occur over previously approved arcs (i.e. arcs with non-zero cost)
- specify that all internally manufactured product must be delivered using one or more internal arcs
- specify that all externally manufactured product must be delivered using one or more external arcs

In addition to these constraints, a side constraint reflects a contractual limit on the number of items delivered from the external sources.

Practice

Go to the **Pasta Production and Delivery** workshop and read the **Product Delivery** step that mirrors the process explained in this lesson.

You can perform this lab using the HTML workshop, or by following the instructions in the workbook. The HTML workshop will give you direct access to OPL documentation pages that can help you with the lab.

Review the problem

Problem description

Once the pasta has been produced, it has to be delivered. The pasta company has internal production sites, and an external production partner with its own site.

- *Products produced in these diverse locations must be delivered to 7 warehouses of the distributor for onward delivery to customers.*
- *A road network links the production sites and the warehouses. The company has established routes it uses, with known costs associated.*
- *The company wants to know how many of each product to ship (i.e. how much to ship from each site) over each existing arc, in order to:*
    - *meet demand*
    - *minimize costs*

This page describes the problem, gives you the data, and lists the steps you will perform next to arrive at the solution. Read through this page carefully, then go on to the next step, Add a network to the project.

***Requirements**:*

- *All produced product will be delivered*
- *Production exactly equals demand*
- *Only routes which have been approved can be used for shipping*
- *Any of the products could potentially be shipped on an approved route.*
- *There is a contractual limit on the amount of outside delivery of any one product to any one warehouse.*

Problem data

In addition to the production data you have already entered into the model, you need the following information:

- *There are 2 production locations, designated* ***`L1`*** *and* ***`L2`.***
- *There are 7 warehouses, designated* ***`W1 - W7`.***
- *The maximum number of an outsourced product to be shipped to any one warehouse is 100.*
- *Approved arcs are shown in the following table:*

		Warehouse						
		W1	W2	W3	W4	W5	W6	W7
Production Site	L1	2	■	2	1	■	■	3
	L2	3	2	■	■	2	1	2
	External	4	■	3	2	4	3	■

Modeling the problem

Note that because products are now manufactured at multiple sites and shipped to multiple warehouses, most of the model data and decision variables will now be specific to either a production location or a warehouse. For example, product demand is now warehouse-specific and resource availability is now location-specific. You will see in the updated work model in the workshop that follows, that we have used tuples to incorporate the new location-specific format of the data.

Steps to the solution

- *Add the distribution side of the network to the model - declare:*
    - *the production locations*
    - *the warehouses*
    - *the approved arcs with their shipping costs (unapproved arcs have a shipping cost of 0)*
- *Declare other data and decision variables:*
    - *The maximum outside delivery of any product to any warehouse*
    - *The amount of internally produced product to deliver from each location to each warehouse*
    - *The amount of externally produced product to deliver to each warehouse*
- *Modify the objective function so that it minimizes shipping as well as production costs*
- *Create new constraints that require:*
    - *Only approved shipping routes can be used (i.e. those that have a non-zero cost assigned to them)*
    - *For each product, the total production at a particular location is equal to the sum of deliveries across all valid routes originating at that location*

Modeling the arcs

The representation of the arcs in this model is in two arrays that each represent a table of values, indexed by **`Locations`** and/or **`Warehouses.`**

Here is the array for the internal arcs:

```
internalArc = #[
       L1 : [ 2 , 0 , 2 , 1 , 0 , 0 , 3]
       L2 : [ 3 , 2 , 0 , 0 , 2 , 1 , 2]
             ]#;
```

Each table entry indicates an arc from a production site to a warehouse.

- Any non-zero value represents the cost of shipping on the arc.
- A value of 0 indicates that the arc is not approved for shipping, or that it does not exist.

Now perform the **Add the network to the project** step. Start with the substeps, **Add the distribution side of the network** and **Declare data and decision variables**.

Add the distribution side of the network

1. *Import the* **`deliveryWork`** *project into the OPL Projects Navigator. It is identical to the* **`productSolution`** *project that you worked on earlier, with indications of where to add new lines.*
2. *In the model file, the warehouses and production locations are already declared with the code,*

   ```
   {string} Warehouses = ...;

   {string} Locations = ...;
   ```
3. *Declare the arcs:*
    - *between internal sources and warehouses*
    - *between external sources and warehouses*

*__Hint:__ for each internal source, declare an array indexed by **`Locations`** and **`Warehouses`** and name this array **`internalArc.`** For each external source, declare an array indexed by **`Warehouses`** and name this array **`externalArc`**.*

4. *In the data file, the 7 warehouses and 2 internal production locations are already initialized.*
5. *Initialize the arcs with their cost data using the table in the previous panel. Use named initialization for clarity. Arcs not listed in the table should have values of 0, as shown below.*

```
internalArc = #[
       L1 : [ 2 , 0 , 2 , 1 , 0 , 0 , 3]
       L2 : [ 3 , 2 , 0 , 0 , 2 , 1 , 2]
             ]#;
externalArc = [ 4 , 0 , 3 , 2 , 4 , 3 , 0];
```

Declare data and decision variables

Declare the following data:

- *Maximum allowable outsourced product to be shipped to any warehouse. Name it **`maxOutsideFlow`**.*

Declare 2 new decision variables:

- *The amount of each internal product to deliver from each location to each warehouse. Name it **`insideFlow`***
- *The amount of each external product to deliver to each warehouse. Name it **`outsideFlow`***

  *For the external product, add a limitation that prevents this delivery from exceeding **`maxOutsideFlow.`***

*In the data file, initialize **`maxOutsideFlow`** with the values given in the previous “Product delivery” step.*

Modeling the objective

For the **objective function,** the **`outsideCost`** and **`insideCost`** data are now represented by the following two tuples:

```
tuple outsideCostData {
   key string p;
   float oc;
}
{outsideCostData} outsideCost with p in Products = ...;

tuple insideCostData
{
   key string p;
   key string l;
   float ic;
}
{insideCostData} insideCost with p in Products, l in Locations
 = ...;
```

The production part of the **objective function** remains to minimize manufacturing costs, except that these costs are now specific to each production location:

```
minimize
 sum(<p,l,ic> in insideCost)ic * insideProduction[p][l]
    + sum(<p,oc> in outsideCost)oc * outsideProduction[p])
```

Extra lines are added to minimize shipping costs:

```
    + sum(p in Products, l in Locations, w in 
Warehouses)(internalArc[l][w] * insideFlow[p][l][w])
    + sum(p in Products, w in Warehouses)(externalArc[w] * 
outsideFlow[p][w]);
```

Now return to the workshop and perform the substeps, **Modify the objective function** and **Create new constraints**.

You can perform this lab using the HTML workshop, or by following the instructions in the workbook. The HTML workshop will give you direct access to OPL documentation pages that can help you with the lab.

Modify the objective function

The objective function already minimizes production costs. It needs to also include an expression that takes into account the following:

- *The different costs associated with each authorized arc (**`externalArc`** or **`internalArc`**)*
- *The total costs for both inside and outside deliveries (**`insideFlow[Locations][Warehouses]`** and **`outsideFlow[Warehouses]`**)*

What needs to be minimized is the total of the shipping costs for all the arcs to be used. The total shipping cost of the arcs for a given product can be expressed as the number of items to be shipped on an arc multiplied by the cost per arc (per item). Write an expression that defines this for:

- *Inside production, indexed by product, location and warehouse*
- *Outside production, indexed by product and warehouse*

Create new constraints

Constraints limiting the arcs:

*The model uses the same array to indicate the cost of an arc, or to indicate that the arc is not authorized or does not exist. A non zero value gives the cost of shipping an item on the arc. A zero value indicates that the arc cannot be used. You must write constraints that test for the zero value and prevent such arcs from being assigned. **Hint:** the decision variables **`insideFlow`** and **`outsideFlow`** represent the number of items to be shipped on an arc. A 0 value assigned to one of these variables means the arc is unused.*

Constraints forcing shipment of all product:

One of the requirements is that production equals demand, and that all product be shipped. Write constraints that require that the number of products produced be equal to the number of products shipped, for both inside locations as well as the external site.

*Name these constraints **`insideBalance`** and **`outsideBalance`**.*

*The solution is found in **`<trainingDir>\OPL63.labs\Pasta\Network\solution\deliverySolution`**. Compare your model with it.*

When it is advantageous to do so, the user can specify that OPL use the **Network Simplex Algorithm** to solve problems with a network structure faster. The different optimization algorithms are discussed in more detail later in this training.

Try using the network algorithm in the workshop. Perform the substep, **Use the network algorithm**.

You can perform this lab using the HTML workshop, or by following the instructions in the workbook. The HTML workshop will give you direct access to OPL documentation pages that can help you with the lab.

Use the network algorithm

As mentioned earlier, OPL has specialized network algorithms that can be used when it is advantageous to do so. This problem is an LP problem but the network you just added gives it a network structure, too. To see how the CPLEX network algorithm solves this problem, do the following:

1. *Use your model, or the one in* ***`<trainingDir>\OPL63.labs\Pasta\Network\solution\deliveryWork`****.* *It should now be open in the IDE.*
2. *Run the model and examine the different tabs in the* ***Output*** *window.*
3. *Double click the settings file,* **`delivery.ops`** *in the project window to open it in the workspace.*
4. *Select* ***Mathematical Programming > General*** *and set the* ***Algorithm for continuous problems*** *to* ***Network simplex****.*
5. *Run the model again, and examine the* ***Engine Log*** *output tab. Notice the difference in the information displayed.*
6. *Examine the other output tabs, and compare them to running the problem with the algorithm set to* ***Automatic****.*

You will note that the solution is identical, and no particular advantage is gained by using the network algorithm compared to the automatically chosen simplex algorithm. You can force use of the network algorithm, if you wish, using the procedure you just completed.

Practice

A further refinement of the model is possible by converting its easy-to-read matrix into more efficient **sparse data**.

Continue in the **Pasta Production and Delivery** workshop and perform the **Make the model sparse** step

You can perform this lab using the HTML workshop, or by following the instructions in the workbook. The HTML workshop will give you direct access to OPL documentation pages that can help you with the lab.

Instructor note
This lab goes with the material on sparsity and slicing in lesson 3. It is placed here because it requires the network lesson. This will be corrected in a future edition of this training.

Make the model sparse

Objective

- *Redefine the model to take advantage of sparse data structures.*

Actions

- *Create the "internalLinkData" and "externalLinkData" tuple*
- *Revise the objective function*
- *Clean up the constraints*
- *Initialize the arcs*
- *Test the model*

Reference

sparsity

Problem description

The pasta company has met with a certain level of success, and wants to expand. Everyone in the company is happy except the OR expert, who is going to have to expand the model you have been experimenting with. If the network of production and distribution becomes too big, it may take a very long time to calculate the best mix of production and transportation to keep costs down and guarantee the continued success of the company. When the first mention of expansion is raised in a company meeting, our OR expert sets out trying to find a way to make the model more efficient so it won't slow down as more sites are added.

A look at the data file in the **`deliveryWork`** *project you worked on earlier, shows that the array of arcs contains several zeros:*

```
internalArc = #[
       L1 : [ 2 , 0 , 2 , 1 , 0 , 0 , 3]
       L2 : [ 3 , 2 , 0 , 0 , 2 , 1 , 2]
            ]#;
externalArc = [ 4 , 0 , 3 , 2 , 4 , 3 , 0];
```

A sparse version of this model would free up memory used to hold zero values that are not used in the optimization calculation. The OR specialist decides to convert these arrays into sparse arrays, to make room for new sites and a larger distribution network.

Exercise folder

<trainingDir>\OPL63.Labs\Pasta\Sparsity\work

Create the "internalLinkData" and “externalLinkData” tuples

1. *Import the* ***SparseDeliveryWork*** *project into the OPL Projects Navigator*
2. *Declare a new tuple named "internalLinkData" containing the following data:*
   - ***string location;***
   - ***string warehouse;***
   - ***float cost;***
3. *Declare a new tuple named “externalLinkData” containing the following data:*
   - ***string warehouse;***
   - ***float cost;***
4. *Declare the arc decision variables (**internalArc** and **externalArc**), making sure that any instance includes the **location** and/or **warehouse** structures from the **internalLinkData** and **externalLinkData** tuples.*

   Hint: *Use the* ***with*** *keyword.*

Revise the objective function

The objective function now needs to be changed so that it works with the new data structures.

- *The first set of expressions in the objective function do not need modification, as they refer to structures that have not changed:*

  ```
  sum(<p,l,ic> in insideCost) ic * insideProduction[p][l]

      + sum(<p,oc> in outsideCost)(oc *
  outsideProduction[p])
  ```

- *The next expression depends, in the existing model, on the array structures **internalArc** and **externalArc** as they were originally declared, with zero placeholders where no arc is authorized. You need to change this to take into account the sparse structure you have just declared.*

  Hint: *The cost summation should now be over the arc sets, as opposed to over the locations and/or warehouses.*

Clean up the constraints

*In the current model, the existing constraints **resourceAvailability** and **demandFulfillment** remain unchanged. Modify the other constraints as follows*

- *Rewrite the constraints **insideBalance** and **outsideBalance** to take the new definition of decision variables **insideFlow** and **outsideFlow** into account.*
- *The test for zero values in the arcs (two unnamed constraints) is no longer necessary, as the sparse structure you've created only initializes valid arcs. These constraints are now handled intrinsically in the data structure itself. Remove them.*

Initialize the arcs

Since the arcs are now declared with only their significant elements, they must be initialized in a different manner. You are going to enter information in the data file to initialize the arcs without any zeros. Only existing arcs will be taken into account by the model, saving memory and calculation time.

- *The structures* **`internalArc`** *and* **`externalArc`** *are declared via the* **`with`** *keyword as instances of the* **`internalLinkData`** *and* **`externalLinkData`** *tuples which contain non-zero values for* **`cost`**. *It is unnecessary to test for* **`cost`**, *since zero cost means the arc does not exist. In the data file, initialize these structures with sets of data that define an arc with these three values. For example,*

```
internalArc =
{<"L1", "W1", 2>,
<"L1", "W3", 2>,...etc.
```

- *Continue to declare all the arcs, per the data table in the* ***Product delivery*** *panel.*

This type of data structure allows OPL to take advantage of both sparse data structures (using tuples) and slicing.

Test the model

1. *In the IDE, load your working file for this lab, import or the solution file,*
   <trainingDir>\OPL63.Labs\Pasta\Sparsity\solution\SparseDeliverySolution
2. *Also load the file*
   <trainingDir>\OPL63.Labs\Pasta\Network\solution\deliverySolution
3. *Run the* **`deliverySolution`** *project. Examine the following items in the* ***Output*** *window:*
   - *In the* ***Statistics*** *tab, explore all the information, especially the number of non-zero coefficients and constraints.*
   - *In the* ***Profiler*** *tab, the total time.*
4. *Run the* **`SparseDeliverySolution`** *project.*
5. *Make the same observations as for the* **`deliverySolution`** *project.*

You will note that the sparse model reduces the number of non-zero coefficients from 168 to 105, thanks to the sparse arrays and slicing. The number of constraints is reduced from 55 to 34. The total time will be determined in part by your machine, but you may see a difference that can become significant in a very large model with a huge array and many zero values.

Practice

Scalability and Sparsity
Up to this point, the network model has remained small. In a real business situation, however, the network can quickly become very large, and performance issues can arise when modeling. This lab presents a much larger version of the original network problem, and then shows you how using sparse data in the model saves time and memory.

Once again, in the **Pasta Production and Delivery** workshop. Perform the **Grow the mode** step

You can perform this lab using the HTML workshop, or by following the instructions in the workbook. The HTML workshop will give you direct access to OPL documentation pages that can help you with the lab.

Instructor note
This lab is optional, if you want to explore scalability and go into greater depth about sparsity with your students.

Grow the model

Objective

- *Observe how a very large model benefits from sparsity and slicing techniques*

Actions

- *Examine and run the enlarged model*
- *Examine and run the sparse version of the model*

Reference

sparsity

Problem description

The worst nightmares of our OR specialist have come true. Not only has the company grown, it has grown exponentially, and now has to deliver products to a network of 6 production locations and 40 000 warehouses!

Exercise folder

`<trainingDir>\OPL63.labs\Pasta\Bigger`

Examine and run the enlarged model

- *Import the project **`bigger`** and double click **`bigger.dat`** to load it into the editing area.*
- *Scroll down the data file until you come to the entries for **`internalArc`** and **`externalArc`**.*

  You can see that the model has been artificially enlarged for this workshop to include 6 production locations 40 000 warehouses. Normally, large amounts of data like this will be imported from a database or spreadsheet. This subject is treated in another workshop.

- *Run the model. When it finishes, examine the information in the different output tabs. Pay particular attention to the **Profiler** tab.*

- *Use the **Problem Browser** window to visualize the properties of all the items in the model by category, and to visualize the results more clearly than is shown in the **Solutions** output tab.*
- *Now examine the model file. Note that, even though it uses the same type of tuple structure as the* **`sparseDelivery.mod`** *file does, the model runs very slowly. Take a close look at the elements of this model.*

Examine and run the sparse version of the model

- *Now load the sparse version of the model, the* **`sparseBigger`** *project. Do not close the* **`bigger`** *project.*
- *Run the model and compare the statistics in the output tabs, especially the **Profiler** tab, with the results from running* **`bigger`**. *It should be obvious that this version of the model runs much faster.*
- *Examine the model and data files for the differences from* **`bigger`**.
- *Discuss the effects of the changes in the model with your instructor. Use the **Problem Browser** window to visualize the properties of all the items in the model by category, and to visualize the results more clearly than is shown in the **Solutions** output tab.*

Summary

Review

Delivering a product over a network of roads is an example of a type of model that requires specialized modeling structures. OPL can model a network, and, when it is advantageous, can use a specialized network version of the simplex algorithm to solve it.

The determining characteristic of network models is that they are comprised of nodes and arcs.

You have explored how to model a network structure in OPL using a simple transportation network example.

You have also seen how sparse data representation can be important in network models, and especially when the scale of the model increases dramatically.

Characteristics of network structures:

- Network models are
- How to model a network structure in OPL
- OPL provides a Network Simplex Algorithm that can be invoked when necessary

Lesson 13: Portfolio Optimization with Quadratic Programming

Linear programming deals with a limited but useful subset of mathematical programming. Integer programming, mixed integer programming and piecewise linear programming all derive from linear programming (LP), and OPL provides efficient algorithms to handle all of these. Some problems, however, require the use of **quadratic expressions.** Stock portfolio optimization represents a common type of quadratic problem.

Instructor note
This lesson should last about 1 hour.

This lesson introduces OPL's support for Quadratic Programming (QP) using a portfolio management example.

In OPL, you can use simple quadratic programs or quadratically constrained programs, in combination with continuous, integer, or mixed integer elements.

Quadratic programming and OPL

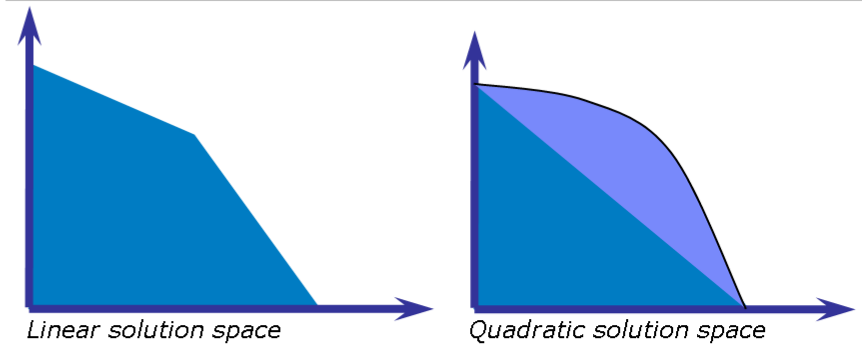
Learning objective
Learn how OPL works with quadratic and quadratically constrained models

Key terms
- Simple Quadratic Programming
- Quadratically Constrained Programming

Linear vs. quadratic

A **linear** optimization program contains a feasible region that is represented, geometrically, by a space bounded by **straight lines.** A **quadratic** optimization program, on the other hand, will have its feasible region bounded by **at least one curved line or surface,** as shown in the following comparison:

Linear and quadratic functions

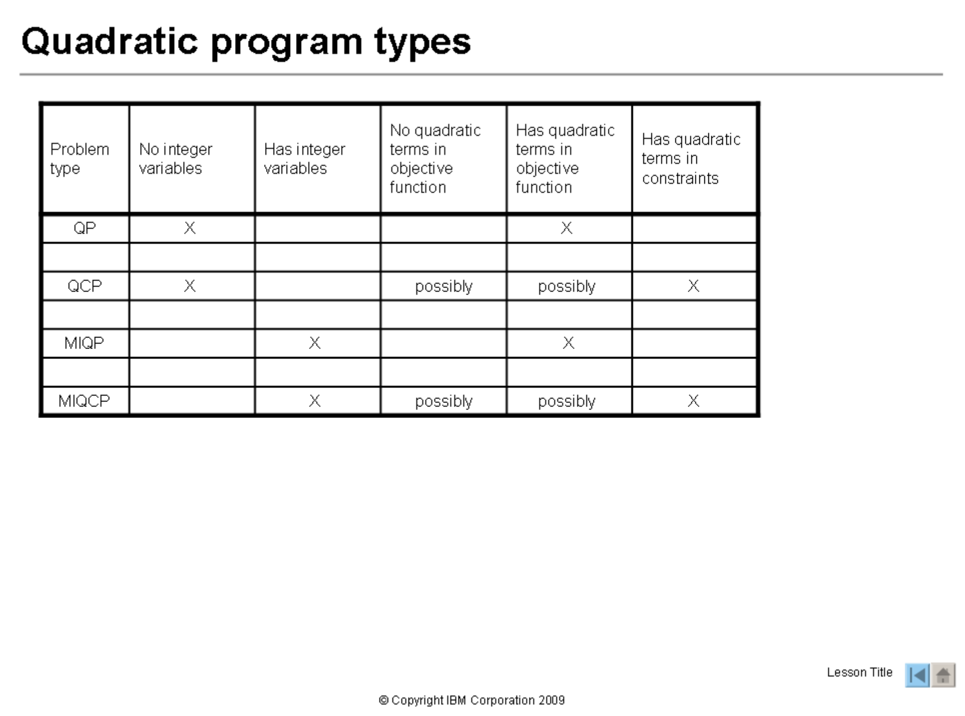


Lesson Title

OPL supports different types of quadratic programming, including:

- Simple QP
- Quadratically-constrained programming (QCP)
- Mixed-integer quadratic programming (MIQP)
- Mixed-integer quadratically-constrained programming (MIQCP)

The following diagram shows the characteristics of each type:



Quadratic program types

Problem type	No integer variables	Has integer variables	No quadratic terms in objective function	Has quadratic terms in objective function	Has quadratic terms in constraints
QP	X			X	
QCP	X		possibly	possibly	X
MIQP		X		X	
MIQCP		X	possibly	possibly	X

Lesson Title

Form of a QP problem

Conventionally, a quadratic program is formulated this way:

Minimize ½ $x^TQx + c^Tx$

subject to Ax <= b

with these bounds lb <= x <= ub

As in other problem formulations, **lb** indicates lower bounds and **ub** upper bounds. The following example shows a QP model:

```
dvar float x[0..2] in 0..40;

maximize
         x[0] + 2 * x[1] + 3 * x[2]
         - 0.5 * ( 33*x[0]*x[0] + 22*x[1]*x[1] +
                   11*x[2]*x[2] - 12*x[0]*x[1] -
                   23*x[1]*x[2] );

subject to {
ct1: - x[0] +     x[1] + x[2] <= 20;
ct2:   x[0] - 3 * x[1] + x[2] <= 30;
}
```

The addition of the quadratic constraint ct3 in the version that follows turns this program into a QCP:

```
dvar float x[0..2] in 0..40;

maximize
         x[0] + 2 * x[1] + 3 * x[2]
         - 0.5 * ( 33*x[0]*x[0] + 22*x[1]*x[1] +
                   11*x[2]*x[2] - 12*x[0]*x[1] -
                   23*x[1]*x[2] );

subject to {
ct1: - x[0] +     x[1] + x[2] <= 20;
ct2:   x[0] - 3 * x[1] + x[2] <= 30;
ct3:   x[0]*x[0] + x[1]*x[1] + x[2]*x[2] <= 1.0;

}
```

OPL can only solve convex quadratic functions.

Practice

Go to the workshop, and perform the **Portfolio Optimization** problem.

You can perform this lab using the HTML workshop, or by following the instructions in the workbook. The HTML workshop will give you direct access to OPL documentation pages that can help you with the lab.

Portfolio optimization

Problem Description

This is a quadratic programming exercise.

Principles of the problem:

- *In order to mitigate risk while ensuring a reasonable level of return, investors purchase a variety of securities and combine these into an investment portfolio. Any given security has an expected return and an associated level of risk (or variance). There is also a tendency for securities to* ***covary****, i.e. to change together with some classes of securities (positive covariance), and in the opposite direction of other classes of securities (negative covariance).*
- *To optimize a portfolio in terms of risk and return, an investor will evaluate the following:*
    - *Sum of expected returns of the securities*
    - *Total variances of the securities*
    - *Covariances of the securities*
- *A portfolio that contains a large number of positively covariant securities is more risky (and potentially more rewarding) than one that contains a mix of positively and negatively covariant securities.*

What to model:

- *Choose securities for the portfolio to improve its return and decrease its volatility.*

    ✓ *As the securities covary with one another, selecting the right mix of stocks can change or even reduce the volatility of the portfolio with the same expected return.*

- *At a given expected rate of return, there is one portfolio which has the lowest risk.*
- *The problem, then, is to write a model that finds the mix of securities that provides the lowest risk for a pre-selected rate of return.*

A quadratic function

- *If you plot each lowest-risk portfolio for each expected rate of return, you will observe that the result is a convex graph, called the efficient frontier.*
- *The risk-return characteristics of a portfolio change in a non-linear fashion, and so,* ***quadratic expressions*** *are needed to model them.*

Exercise folder

`<trainingDir>\OPL63.labs\Portfolio\work`

Write objective and constraints

Objective

- *Use QP principles to create a portfolio optimization model*

Actions

- *Model the problem using QP*
- *Define objective and constraints*
- *Use a logical constraint to limit diversity*

Reference

quadratic programming

Model the problem using QP

- *Import the* **`portfolioWork`** *project and examine the* **`portfolio.mod`** *file.*
- *In this model, most of the problem is already defined. It is up to you to write the objective and constraints.*
- *The data elements are:*

```
{string} Investments = ...;
float Return[Investments] = ...;
float Covariance[Investments][Investments] = ...;
float Wealth = ...;
float goalReturn = ...;
range float FloatRange = 0.0..Wealth;
```

- *The decision variables are:*

```
dvar float  allocation[Investments] in FloatRange;
```

Define objective and constraints

- *The objective is to minimize the portfolio risk (variance).*
  The covariance of returns of stocks **`i`** *and* **`j`** *is defined as the allocation of the portfolio to stock* **`i`** *times the allocation of the portfolio to stock* **`j`** *times the covariance between* **`i`** *and* **`j`**.
- *Write the constraints*
  - *Allocate All Wealth*
  - *Meet Total Return Minimum*
    Return is defined as the sum of the allocation of the portfolio to stock **`i`** *times the expected return of stock* **`i`**.
- *Run your solution and debug it if necessary.*

Use a logical constraint to limit diversity

If you run the solution file **`<trainingDir>\OPL63.labs\Portfolio\solution\portfolioSolution`**, *you will note that every possible security has an allocation assigned to it. In many cases, this type of diversity of investment is desirable, but some investors' objectives may include a limit on the number of different stocks in their portfolio.*

To do this, it might seem logical to simply use a linear constraint: count the number of **`allocations`** *(using the* **`card`** *function) and ensure that this number is always greater than or equal to a data element we set as the diversity limit (call it* **`maxSecurities`***). However,* **`card`** *only works over sets, and* **`allocations`** *is a decision variable, so another way has to be found, using a logical constraint.*

1. *Copy the* **`portfolioSolution`** *project and paste it as***`<trainingDir>\OPL63.labs\Portfolio\work\portfolioLimitedWork`**

You must do this from inside the OPL Projects Navigator. Do not attempt to copy and paste the project directory in the file system, you will not be able to import the copy.

2. *Add an integer data declaration to create the data element* **`maxSecurities.`** *This can be initialized internally in the model or from the* **`.dat`** *file, as you wish. Set its value to 5.*
3. *Define a logical constraint that says that the number of non-zero* **`allocations`** *must be equal to or less than* **`maxSecurities.`**

To set the maximum for non-zero applications equal to **`maxSecurities,`** *use a formula that tests for the number of allocations set to 0, and constrains the model to limit that number to be at least equal to or greater than the difference between the total number of securities under consideration, and* **`maxSecurities`** *(Remember, strict inequalities are not permitted in OPL.).*

4. *Run your solution and debug it if necessary.*

A different logical constraint: investment percentage

As already pointed out, investors' objectives can be very different, so our model needs to be flexible enough to be reused with a variety of investor needs. Simply by changing our logical constraint, for example, we can adjust the model to require that any investment represent, at minimum, 6% of the overall portfolio value.

1. *Copy the* **`portfolioLimitedWork`** *project and paste it as***`<trainingDir>\OPL63.labs\Portfolio\work\portfolioLimited2Work`**

You must do this from inside the OPL Projects Navigator. Do not attempt to copy and paste the project directory in the file system, you will not be able to import the copy.

2. *Replace the integer data declaration* **`maxSecurities.`** *with a float value called* **`minAllocation`**. *Initialize this value from the* **`.dat`** *file as 0.6.*
3. *Replace the logical constraint* **`maxStock`** *with one called* **`minInvestment`**. *This time, it must require that any investment, at minimum, must be greater than or equal to the value of* **`minAllocation`**.
4. *Run your solution and debug it if necessary.*
5. *Run the project and compare the results with the results from the* **`portfolioSolution`** *project.*
6. *How many allocations are there? Try changing the value of* **`minAllocation`** *and running the problem again. Compare the different results*

Solutions

Actions

- *Define objective and constraints*
- *Use a logical constraint to limit diversity*

Define objective and constraints

The solution file can be found in
`<trainingDir>\OPL63.labs\Portfolio\solution\portfolioSolution.`

```
/******************************
 * Objective and constraints *
 *****************************/

minimize
     (sum(i,j in Investments)
Covariance[i][j]*allocation[i]*allocation[j]);

subject to {
   // sum of allocations equals amount to be invested
   allocate: (sum (i in Investments) (allocation[i])) ==
Wealth;

   // achieve a minimum return on investment
  minReturn: (sum(i in Investments) Return[i]*allocation[i])
 >= goalReturn;
```

Use a logical constraint to limit diversity

The solution file can be found in
<trainingDir>\OPL63.labs\Portfolio\solution\portfolioLimitedSolution.
Note the additional declaration of the **maxSecurities** *variable and the* **maxStock** *logical constraint.*

```
/*******
* Data *
*******/

{string} Investments = ...;
float Return[Investments] = ...;
float Covariance[Investments][Investments] = ...;
float Wealth = ...;
float goalReturn = ...;

// Variable to set the maximum number of securities in the
portfolio
int maxSecurities = ...;

range float FloatRange = 0.0..Wealth;

/*********************
 * Decision variables *
 *********************/

dvar float  allocation[Investments] in FloatRange;  //
Investment Level

/*************************************
 * Objective function and constraints *
 *************************************/

minimize
     (sum(i,j in Investments)
Covariance[i][j]*allocation[i]*allocation[j]);

subject to {
   // sum of allocations equals amount to be invested
   allocate: (sum (i in Investments) (allocation[i])) ==
Wealth;
```

```
   // achieve a minimum return on investment
  minReturn: (sum(i in Investments) Return[i]*allocation[i])
 >= goalReturn;

   // use a logical constraint to limit the number of
different securities we can use
   maxStock: sum(i in Investments)
(allocation[i]==0)>=card(Investments)-maxSecurities;
}
```

- *Run the project and compare the results with the results from the* **`portfolioSolution`** *project.*
- *How many allocations are there?*
- *Try changing the value of* **`maxSecurities`** *and running the problem again. Compare the different results.*

A different logical constraint: investment percentage

The solution file can be found in
<trainingDir>\OPL63.labs\Portfolio\solution\portfolioLimited2Solution.
Note the additional declaration of **`minAllocation`** *and the* **`minInvestment`** *constraint.*

```
/*******
* Data *
*******/

{string} Investments = ...;
float Return[Investments] = ...;
float Covariance[Investments][Investments] = ...;
float Wealth = ...;
float goalReturn = ...;

// data to set the minimum allocation percentage
float minAllocation=...;

range float FloatRange = 0.0..Wealth;

/**********************
 * Decision variables *
 **********************/

dvar float  allocation[Investments] in FloatRange;  //
Investment Level

/**************************************
 * Objective function and constraints *
 **************************************/

minimize
    (sum(i,j in Investments)
Covariance[i][j]*allocation[i]*allocation[j]);

subject to {
   // sum of allocations equals amount to be invested
   allocate: (sum (i in Investments) (allocation[i])) ==
Wealth;
```

```
   // achieve a minimum return on investment
  minReturn: (sum(i in Investments) Return[i]*allocation[i])
 >= goalReturn;

   // use a logical constraint to require that any investment
 represent, at minimum, 6% of the total portfolio value
   minInvestment: forall(i in Investments)
(allocation[i]!=0)=>(allocation[i]>=minAllocation*Wealth);

}
```

- *Run the project and compare the results with the results from* **`portfolioSolution`**.
- *How many allocations are there? Try changing the value of* **`minAllocation`** *and running the problem again. Compare the different results*

Summary

Review

In this lesson, you learned:

- Some processes that you want to optimize (for example, stock portfolio management or chemical interactions) cannot be expressed by linear functions. These require quadratic functions to be modeled.
- OPL supports different types of quadratic programming:
    - Simple QP
    - Quadratically-constrained programming (QCP)
    - Mixed-integer quadratic programming (MIQP)
    - Mixed-integer quadratically-constrained programming (MIQCP)

You have also had hands-on experience writing a model that uses these techniques to optimize a portfolio of investments.

Lesson 14: From Model to Application - The ODM Connection

OPL is one of the tools that constitutes the IBM® ILOG® Optimization Suite, a tightly integrated decision-support system that provides a complete solution for the development and deployment of optimization-based planning and scheduling applications.

IBM ILOG Optimization Decision Manager (ODM) works together with OPL to provide an environment for prototyping and development of end-user applications

Instructor note
This lesson is optional. If performed, it should last about 2 hours, including the practices. It is not intended to train the students how to use ODM nor is it a complete course in application development. Its primary purpose is to introduce the OPL-ODM connection and demonstrate the synergy of using these two tools together.

In this lesson, you will learn:

- How ODM functions as a natural extension of OPL
- How to generate an application from your model quickly and easily
- How to customize an application so that business users have easy access to key information

This lesson gives only a quick overview of the power and potential of OPL-ODM integration. For more information, contact your IBM representative.

What is an ODM application?

Learning objective
Understand how ODM generates an interactive end-user application from an OPL model

Key terms
- Optimization Decision Manager (ODM)
- goal
- requirement
- scenario
- "what-if" analysis

About IBM ILOG ODM

You can generate standalone applications based on OPL models in less than a minute, using IBM ILOG Optimization Decision Manager (ODM). ODM is both a tool for application development and a runtime environment. It puts all the power of OPL models in the hands of business users and other non specialists who must make decisions based on your models.

When both OPL and ODM are installed, you can use OPL to generate and customize applications that permit business decision makers to create different scenarios using the same model. This allows users to:

- Perform 'what if' analysis on optimization solutions - generating decision scenarios by changing input data, requirements, cost and yield assumptions, goals, and business rules
- Easily compare these decision scenarios
- Use ODM's built-in charting functions to visualize and analyze input and solution data
- Export the decision scenario result sets to Excel for printing and further analysis

About IBM ILOG ODM applications

An application generated in IBM ILOG Optimization Decision Manager (ODM) is a representation of an OPL model that is quickly, easily, and intuitively accessible to business users. It does this in two ways:

- It represents the OPL model in terms of necessary input data, results, and goals, in a language that is familiar to business users.
- It provides the business user with easy access to the above-mentioned data, and allows the user to change data and/or constraints to make different scenarios, which can be saved separately ("what-if" analysis).

In ODM, decision expressions in the objective function are mapped to **goals** and OPL constraints are mapped to **requirements**. The requirements can be easily relaxed or further constrained by the user, or their priorities can be changed to make them more or less important. If there is an infeasibility, ODM reports the requirements that conflict with the current data, and helps the user either to redefine the problem requirements or to change the data.

Because ODM allows users to easily create and compare multiple scenarios, they can explore alternatives and their impact on costs, revenues, or other goals in those different scenarios.

ODM applications can be deployed as final applications to your end users. Using Java^{TM}, you can extend the default ODM application by adding your own custom views, more adapted to your business problem. You can also add specific data sources, if you need more than SQL queries, Excel files or text files to get to your data.

Model limitations

Some limitations apply to models converted to ODM applications:

- The models should be LP, MP, MIP, or CP. It is possible to use quadratic expressions, but with some limitations.

- Model size can have an impact on ODM performance. For models that are solved interactively on a local PC in real time, ODM works best for models that are solved in less than 10 minutes. Models which are solved externally via a batch process, however, can be larger.

Instructor note
The most important point to convey to the students with regard to this limit is that models that are appropriate for real time interactive solution with ODM should be solvable in a short enough time to insure interactivity on the part of the user. Users should be able to click Solve and get a result fairly quickly, so that they can see their results without waiting for them.
Batch mode solutions provide a facility to let solves run in the background while the user does other tasks. When the batch has finished running, the results can be viewed in ODM. Batch mode is not treated in this training, but you can introduce it if you wish.

ODM architecture

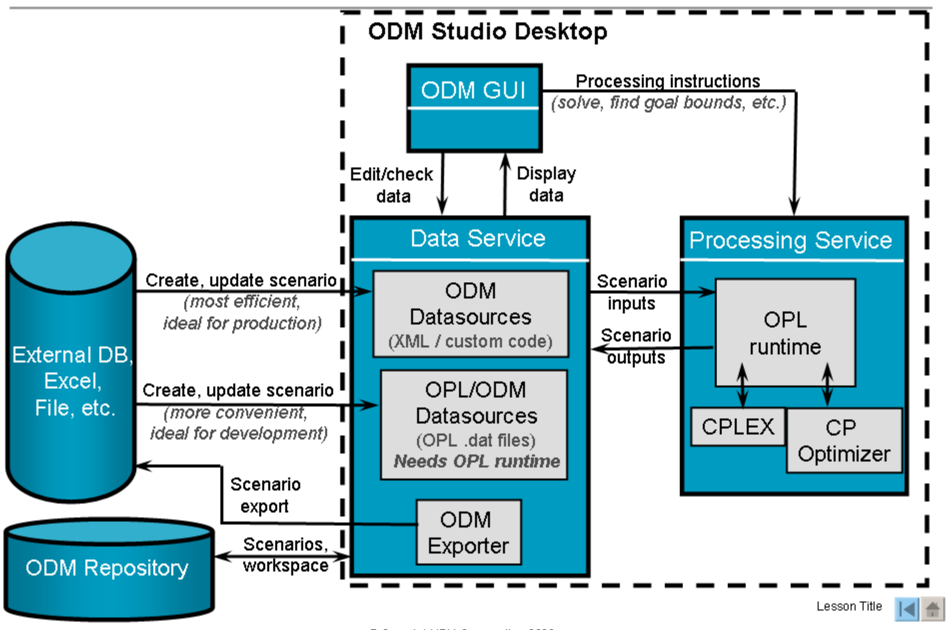
Learning objective
Understand the basic ODM architecture in terms of its components and how they relate to each other.

Key terms
- solver engine
- CPLEX
- CP Optimizer
- OPL Runtime
- ODM Studio
- scenario
- relational model
- ODM repository
- ODM view

Inside IBM ILOG ODM

The following diagram shows the basic ODM runtime architecture for a single-user desktop:

ODM runtime architecture: Single-user desktop



© Copyright IBM Corporation 2009

The three main components of the ODM Studio Desktop are as follows:

- The ODM GUI: This is where the user works with scenarios.
- The processing service, where scenarios are solved, consisting of:
  - OPL runtime
  - CPLEX® solver engine: Used to solve LPs or MIPs. CPLEX can also be used to solve some QP problems, but with certain limitations.
  - CP Optimizer solver engine: Used to solve problems that require constraint programming, especially scheduling problems.

- The data service, consisting of:
    - ODM data sources (XML/custom code)
    - OPL/ODM data sources (OPL .dat files)
    - ODM data exporter

Instructor note
There are limitations on quadratic expressions in ODM for the current release: If models use quadratic expressions in objective functions (QP or MIQP), the quadratic objectives are not monitored and not seen in the solve progress panel in ODM. You should use caution when making quadratic constraints relaxable, as in certain cases this might lead to non-positive semi-definite problems, which CPLEX cannot handle.

These three components of the ODM Studio Desktop talk to each other in the following way:

- The ODM GUI
    - Sends processing instructions to the processing service, for example to solve a scenario.
    - Sends instructions to edit or check data to the data service.
- The data service
    - Sends data for display to the ODM GUI.
    - Sends/receives scenario data to/from the processing service.
- The processing service, using OPL runtime
    - Converts the OPL model and data to a CPLEX or CP Optimizer model.
    - After solving the model, sends the solution data back to the data service

The following databases are part of the runtime architecture, but external to the ODM Studio Desktop:

- The ODM repository
    - Stores scenario data.
    - Sends/receives scenario and workspace data to/form the ODM data service.
- Other external databases, excel, files, etc.
    - Send data to the ODM data service to create or update scenarios.
    - Receive scenario data exported by the ODM data exporter.

Note the following important distinctions between OPL/ODM data sources (OPL .dat files), and pure ODM data sources (XML or custom code):

OPL .dat	**XML or custom code**
2-step data transfer (data source to OPL, OPL to ODM)	1-step data transfer (data source to ODM)
More convenient	More efficient
Needs OPL runtime	Does not need OPL runtime
Ideal for development	Ideal for production

Generating a basic ODM application

Learning objective
Learn how to generate and customize a basic ODM application from an OPL model.

Key terms
- ODM Application Generation Wizard
- Scenario Explorer
- default mapping
- goals
- goal view

Before generating an ODM application, you need:

- An OPL model
- Specifications (mappings) that describe how items in ODM (rules, goals, requirements, etc.) will be linked to items in OPL (constraints, decision expressions, decision variables, etc.)
- A list of the data views you want to display in ODM
- Optionally, a list of data sources for ODM
- Optionally, a list of custom views for ODM (to display data in a format that is not proposed by default: for example, maps, charts, diagrams, Gantt charts)

The initial ODM application is configured in OPL. It can then be customized further directly using the ODM editors, by editing XML files generated by the OPL connector, or by adding custom Java^{TM} code.

Generating an ODM application is a quick and easy process. As a demonstration, you are now going to generate a simple ODM application from an OPL model, together with your instructor. In the labs for this training you will work with a single application used to address a Unit Commitment Problem (UCP).

Power generation - the unit commitment problem

The purpose of the UCP is to plan the operation of a number of power generators over a given time horizon with a known load (demand) profile. The data used by this application includes:

- The demand (load) forecast over a given planning horizon. The load levels correspond to a large state, region or province, or 2-3 smaller ones, so our fictitious utility company is a fairly large supplier.
- A set of generators (units) with a number of characteristics, namely initial production level, minimum and maximum generation levels, minimum up and down time, maximum ramp up and ramp down rate, a startup fuel consumption coefficient, and a fuel consumption formula.

For each time period, the decision variables include:

- Status of each unit (non-exclusive):
  - In use
  - Being turned on
  - Being turned off
- The production level for each unit.

The constraints involved in the decision making process are as follows:

- The forecasted load must be satisfied.
- A unit that is in use must operate between defined minimum and maximum power generation levels.
- Restrictions on the production load to which a unit can ramp up or down during consecutive time periods must be respected.
- Once a unit turns off, a minimum down time must be respected.
- Once a unit turns on, a minimum up time must be respected.
- The total operating cost, consisting of the fuel cost and startup cost for each unit, must be minimized.

In addition, the application allows the user to specify a number of rules regarding:

- Unit maintenance
- Maximum hours of usage per day
- Specific time periods during which a given unit must be either on or off
- Spinning reserve requirement. This refers to a production reserve to deal with differences between the actual and forecasted load levels.

The ODM application generated from this model also includes a number of Key Performance Indicators (KPIs), such as utilization and alternative cost measures.

In the next section, you'll get familiar with the basic model for this problem. This basic model has an objective to minimize costs, while meeting demand and satisfying the constraints on the operating conditions of each generator. Later on you'll see how the ODM application can be customized to deal with special situations, such as:

- Suppose you need to perform maintenance or repairs on a given generator. You can create a scenario to determine the best time to do this.
- Actual load values are more than likely to vary from the forecasts you are using. It will be necessary, then, to reserve a certain amount of capacity ("Spinning Reserve") at all times to cover any extra demand.

Practice

Getting to know the UCP model

You are now going to perform a lab in which you will:

- Get familiar with the UCP model in OPL
- Generate a default ODM application
- Get familiar with some basic elements of an ODM application

You can perform this lab using the HTML workshop, or by following the instructions (in *italics*) in the workbook. The HTML workshop will give you direct access to ODM documentation pages that can help you with the lab. The first step is to **Generate a default ODM application**.

Exercise directories

- **`<TrainingDir>\OPL63.labs\default`**
- **`<TrainingDir>\OPL63.labs\BaseODMapp`**

Study the exercise directory

1. *Open the exercise directory* ***`<TrainingDir>\OPL63.labs\default\ucpDefault`*** *and spend a minute or two familiarizing yourself with its contents:*
    - *There are five OPL files:*
        - ***`ucp.mod`*** *is the OPL model file.*
        - ***`ucp.dat`*** *contains the data.*
        - ***`ucp.ops`*** *is the settings file that can be used to fine-tune the behavior of the optimization engine or to modify other default setings.*
        - ***`.project`*** *is the project description file.*
        - ***`.oplproject`*** *is the opl project description file which maintains run configurations.*
    - *There are two MS Excel files.*
        - ***`unitData.xls`*** *contains data for each generator.*
        - ***`loadData.xls`*** *contains load data for each time period.*

    These data files contain input data to the OPL model. The specific data read into the model and the order in which it is read are specified in the ***`ucp.dat`*** *file.*

Open the project in ODM Enterprise

IMPORTANT: Only follow this step if you are using the ODM Enterprise IDE. For the desktop version of ODM, you'll use the OPL IDE to open projects.

1. *Launch* ***Eclipse for ODM Enterprise*** *from the Windows® Start menu by choosing* ***Start > All Programs > IBM ILOG > IBM ILOG ODM Enterprise 6.3 > Developer > Eclipse for ODM Enterprise****.*
2. *In the main menu bar, choose* ***Window > Open Perspective > Other...****. In the* ***Open Perspective*** *window, choose OPL. The OPL perspective you now see on the screen looks the same as the OPL IDE you would see if you were working with OPL as a standalone product.*
3. *In the main menu bar, select* ***File >Import****. In the* ***Import*** *window, expand* ***OPL*** *and select* ***Existing OPL 6.x projects****.*
4. *Choose* ***Next****, and in the* ***Select root directory*** *field navigate to* ***`<TrainingDir>\OPL63.labs\default\ucpDefault`*** *directory. Make sure that the* ***Copy projects into workspace*** *checkbox is* ***NOT*** *checked. Click* ***OK****. Note that the* ***`ucpDefault`*** *project is the file that contains and manages all the elements of the project.*

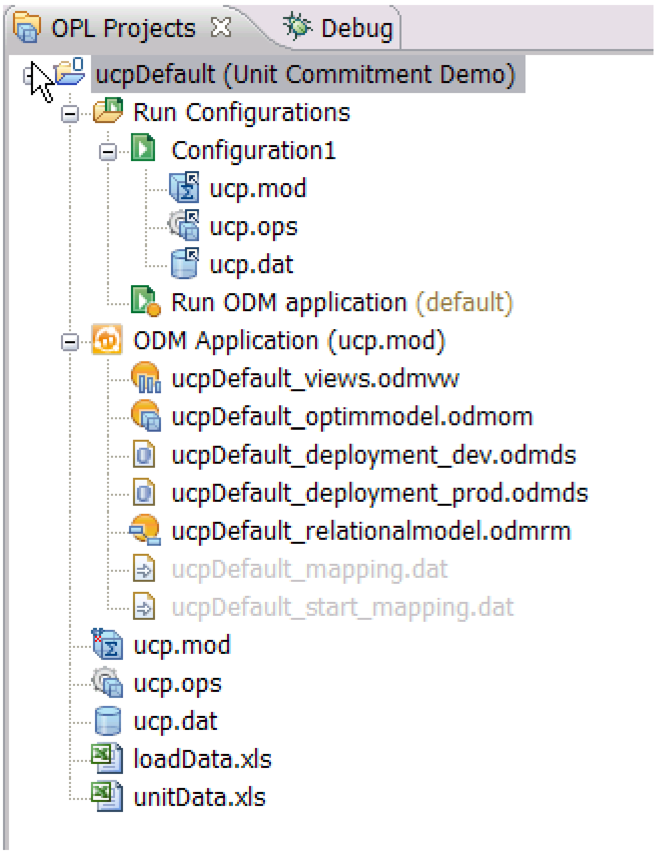
Generate a default ODM application

1. *In the OPL Projects Navigator (the window in the top left corner of the IDE), expand the* **`ucpDefault`** *project, and look at the contents of the* **`.mod`** *and* **`.dat`** *files. You don't need to fully understand the content, but see if you can relate it to the problem description given before.*
2. *In the* ***OPL Projects Navigator****, right-click the name of the project (in this case,* ***ucpDefault (Unit Commitment Demo))*** *and choose* ***Generate ODM Application*** *from the context menu. The* ***ODM Application Generation Wizard*** *opens.*
3. *Two options are presented:*
   - ***Generate a default ODM application*** *creates an ODM application with a full representation of model and data from the OPL model.*
   - ***Generate an empty ODM application*** *generates an empty ODM application with no data or model information.*

   Select ***Generate a default ODM application*** *and click the* ***Next*** *button. At this point, you could skip the next three steps, if you don't need them, by clicking* ***Finish****. In this case, however, we will have a look at the screens to see what they do.*
4. *The next screen asks you to select model and data files to use in the ODM application. Since this project contains only one of each, they are already selected:*
   - ***ucp.mod*** *is the only choice in the drop-down list box.*
   - ***ucp.dat*** *is the only file shown in the data file list.*

   You can have more than one of either type of file in the project, and you then have to select which you want to use in the ODM application. Click ***Next****.*
5. *The next screen asks for the name of the ODM application. Accept* ***ucpDefault*** *as the name by clicking the* ***Next*** *button.*
6. *The final screen presents a summary of the options you have chosen. You could correct previous actions by clicking the* ***Back*** *button, but at this point, accept them by clicking* ***Finish****.*

The application is generated, and you will now see 2 new entries in the OPL Projects Navigator tree:



```
ucpDefault (Unit Commitment Demo)
  Run Configurations
    Configuration1
      ucp.mod
      ucp.ops
      ucp.dat
    Run ODM application (default)
  ODM Application (ucp.mod)
    ucpDefault_views.odmvw
    ucpDefault_optimmodel.odmom
    ucpDefault_deployment_dev.odmds
    ucpDefault_deployment_prod.odmds
    ucpDefault_relationalmodel.odmrm
    ucpDefault_mapping.dat
    ucpDefault_start_mapping.dat
  ucp.mod
  ucp.ops
  ucp.dat
  loadData.xls
  unitData.xls
```

*The entry **ODM Application** expands to seven items. Five of these are used to customize the ODM application, and the other two that are greyed out (and cannot be edited by the OPL user) define the mapping between OPL and ODM. There is also a new run configuration, called **Run ODM application**. This is a special run configuration that launches the ODM application (as opposed to running the model in OPL).*

Solve options

ODM allows you to control how the solve algorithm will behave when a solve is launched from ODM Studio. This behavior differs slightly from when the solve is launched through OPL.

Click **ODM Application > ucpDefault_optimmodel.odmom** in the OPL Projects Navigator, and select the **Model Options** tab.

Under the **Solve Algorithm** heading, there are three options:

- **Solve**: A good option if you feel certain that there will be no constraint violations. This will speed up processing, but if the problem is in fact infeasible, no solution will be found.
- **Solve First**: The optimization engine builds a requirements tree, but then attempts to solve the problem using the **`Solve`** algorithm. If that fails, it starts a new solve using the **`solveAnyway`** algorithm.
- **Solve Anyway** (the default): Build on the CPLEX® **`feasOpt`** algorithm which finds minimal constraint relaxations when a model is infeasible.

Remember that relaxable constraints correspond to requirements in ODM – if a constraint is relaxed by Solve Anyway, it will appear as a requirement in ODM.

For CP problems, only the **Solve** option is available.

More on Solve Anyway

Inside CPLEX the **Solve Anyway** algorithm performs a loop on top of **`feasopt`**.

Solve Anyway systematically adds constraints in the "relaxable pool" of constraints, starting with those that have the lowest priorities, until a feasible solution is found.

When a feasible solution is found, **Solve Anyway** will try to optimize the objective function.

ODM users can set priorities on constraint relaxation.

The Solve Queue

From ODM3.0 onwards, solve operations can run in the background, while users continue to work with scenarios in ODM Studio in the foreground.

- You can submit multiple solve operations to run in the Solve Queue.
- Scenarios are solved in the order submitted.
- You can check on the solve progress or cancel a solve at any time.
- You can abort a solve process that is not active yet, but waiting in the queue.

Return to **Step 1** of the **Introduction to the Unit Commitment Problem** workshop and perform the substeps, **Launch the ODM application** and **Create and solve the default scenario**.

Launch the ODM application

- *In the **OPL Projects Navigator**, the **Run ODM application** run configuration has automatically been set as the default. This is a special run configuration that launches the ODM application. Right-click on this run configuration and choose **Run this** from the context menu. This will launch ODM Studio (ODM's graphical user interface) with the **ucpDefault** application loaded.*

Create and solve the default scenario

1. *Look at the ODM Studio window. Although the **ucpDefault** application is loaded, you cannot see any data yet. This is because no scenarios have been created or imported yet. In the ODM Studio main menu bar, select **File > New > New Default Scenario.** A scenario will be created containing the data and model information as configured in OPL.*
2. *In the **Scenarios Overview** window, right click **New Default Scenario** and select **Rename** from the context menu. Rename the scenario **ucp Baseline** and press **`<enter>`.***
3. *In ODM, click the **Solve** button (third from the left) in the main toolbar to display the **Solve Progress** window and solve the problem. When the solve completes, click the **Close** button.*

ODM goals

In ODM, goals are optimization criteria. ODM optimizes the sum of different goals as a function of assigned weights for each goal. In the ODM **Goals view** you can:

- See the value of goals in the current solution

- Specify a goal's weight
- See the goal breakdown
- Search the goal bounds
- Add constraints to its value

To create an ODM goal, the OPL objective (an expression to maximize or minimize) must be expressed as a weighted sum of one or more OPL decision expressions (**`dexpr`**s). Each **`dexpr`** element of the objective becomes a goal in ODM.

While OPL does not require the use of **`dexpr`** expressions in the objective function, these are required for the creation of ODM goals.

Each goal can have a weight or importance factor, which in OPL is the coefficient multiplying the **`dexpr`** in the minimize or maximize statement. The weight assigned to each of the **`dexpr`** expressions is used as an **importance factor** in ODM when calculating the goal total.

Note the following important rules when writing an OPL objective for an ODM application:

- Weights must be explicitly set in the objective function and cannot be derived from another data element. For those already familiar with OPL syntax, the following is not allowed:
  ```
  float coefficient[terms] = [term : coeff |
  <term,coeff> in objectiveCoefficients]
  ```
  Here, a **`float`** array called **`coefficient`** is indexed by the objective **`term`**s. The value of each array element (each coefficient for each objective term) is derived from another data element, namely a tuple **`<term, coeff>`** in the set **`objectiveCoefficients`**.
- The objective must not contain parentheses.

Perform the rest of the substeps in the **Generate a default ODM application** step of the **Introduction to the Unit Commitment Problem** workshop.

Examine the goals

1. *In the **Scenario Explorer** window of ODM, expand the **Analysis** entry and click **Goals** to display the Goals window. In this example, two decision expressions have been converted to ODM goals as part of the default mapping performed by the **ODM Application Generation Wizard**:*
   - *FuelCost*
   - *StartUpCost*

   *Examine how these are described in the OPL model file and how they appear in the **Goals** window, to understand the relationship.*
2. *Select **FuelCost** in the **Goal Name** column of the upper pane. Then, in the lower pane of this window, click the **+** sign beside **FuelCost** to expand this item and see the values assigned to each fuel type for this decision expression. Do the same for **StartUpCost**.*

Compare the model representation in OPL and ODM

1. *In the **Scenario Explorer** window, click **Analysis > Requirements**.*

   *No ODM requirements are displayed in the **Requirements** window, because no requirements needed to be relaxed to find a solution.*

2. *In **Scenario Explorer**, expand the **Input Data** entry and click on each of the tables to open them in the main window.*
   - *Notice the column names in each of the tables – these are the default mappings of the input data definitions in the OPL model. These default names may be changed later.*
   - *Notice also that the actual values in the **Input Data** tables, taken from the **ucp.dat** file, are editable. Although it is not used here, the Scenario data checker feature in ODM enables a developer to embed business-specific data validation rules, which are executed when the business user solves a scenario and which report errors and warnings before solving the scenario.*
3. *Expand the **Solution** entry in **Scenario Explorer** and choose the **production** table. The column names in this table show the default mapping from the OPL model. The numbers are generated when the model is solved.*
4. *Make sure you understand how the data shown in ODM are mapped to the declarations in the OPL model.*

Examine the work directory

Now open the exercise directory ***`<TrainingDir>\OPL63.labs\Default\ucpDefault`*** *and look at the files that have been generated there.*

There are four new editable XML files:

- ***`ucpDefault_views.odmvw`*** *– contains the description of how the input and solution data is displayed in the different ODM views.*
- ***`ucpDefault_optimmodel.odmom`*** *– contains descriptions of the components of the optimization model, including requirements, goals and rules.*
- ***`ucpDefault_deployment_dev.odmds`*** *– contains descriptions of the data sources and deployment configurations used by the ODM application in development mode.*
- ***`ucpDefault_deployment_prod.odmds`*** *– contains descriptions of the data sources and deployment configurations used by the ODM application in production mode.*

*These files can be edited in OPL by clicking their entries under **ODM Application** in the OPL Projects Navigator. They can also be edited directly using a text editor or an XML editor.*

A distinction is made in ODM between development mode and production mode to allow you to create different data sources and run configurations for each one. The ***`...deployment_dev.odmds`*** *file is used during development, while the* ***`...deployment_prod.odmds`*** *file is intended for production.*

The entry, ODM Application (ucp.mod), in the OPL Projects Navigator contains a set of files that you can also see in the exercise directory:

- ***`ucpDefault_relationalmodel.odmrm`***
- ***`ucpDefault_mapping.dat`***
- ***`ucpDefault_start_mapping.dat`***

The latter two files are grayed out in OPL, and you cannot edit them from the IDE. They define tables in the ODM relational model, and how the ODM data in those tables are mapped to the OPL data. These files are not usually edited by developers, though for some tasks you may need to access them.

When you have finished examining the files, close the ODM application. Select the **`ucpDefault`** *project from the OPL Projects Navigator, and delete it from OPL by right-clicking and selecting* ***Delete****. When the* ***Confirm Project Delete*** *window appears, make sure* ***Delete project contents on disk*** *is* ***NOT*** *selected before deleting, otherwise the project will also be deleted from your computer.*

Creating an ODM application from a CP Scheduling model

In this practice, the instructor will demonstrate:

- How to generate an ODM application from an OPL model that uses constraint programming (CP) to solve a scheduling problem.
- The appearance of the default application.
- How to create a chart within ODM to combine data from two different ODM tables into one graphical representation to better visualize the solution.

You do not have to perform the steps of this demo yourself

Problem Description

This problem is used to demonstrate how to generate an ODM application from an OPL model that uses CP Optimizer (instead of CPLEX®) to solve a scheduling problem. You do not have to perform the steps of this demo yourself, but the project is available to you in **`<TrainingDir>\OPL63.labs\Scheduling\sched_wood_for_ODM\work\sched_wood_for_odmWork`** if you want to experiment with it after the completion of the course.

The wood cutting example involves a machine in a wood cutting factory that cuts stands (processed portions of log) into chips. Each stand has a certain length, diameter and species. The machine can cut a limited number of stands at a time, with some restriction on the sum of the diameters that it can accept. Only one species of wood can be processed at the same time. Finally, each stand has fixed delivery dates and a processing status — either 'standard' or 'rush.' Any delay on 'rush' stands will incur a penalty. The objective is to minimize the total cost of operating and delay, while satisfying the following constraints:

- The truck fleet that carries stands to machines for processing is limited.
- The machine is a discrete resource with capacity specified in terms of the number of stands that can be cut at the same time.
- In addition to the diameter constraint, only a limited number of stands can be loaded at the same time.
- To express that only one species can be cut at the same time a **`state`** resource is used. At any given time, this resource indicates the state in terms of which species the machine can cut at a given moment.

One of the aspects of the problem that the user wants to monitor is that the end times for each stand in the solution schedule meet the established due dates for delivery. This information is displayed as data within OPL, but might be difficult for a business user to envision. So in this demonstration the instructor will use ODM's built-in charting capability to create a visual representation of this data.

Generating the Wood Cutting application

Watch as the instructor demonstrates this process.

Instructor note
Use the following procedure for this demo:
Generating the ODM application

1. Open the **`<TrainingDir>\OPL60.labs\Scheduling\sched_wood_for_odm`** project.
2. Double-click on **sched_wood.mod** to show the students that this is a CP model.
3. Choose **Generate ODM Application** from the File menu. On the opening screen, click **Finish**, explaining that in this model there is

only one model and data file, and so there is no need to go through the individual screens that they saw in their first practice.

4. Point out the additional objects added in the OPL Projects panel, and then run the ODM application.
5. Solve the problem, and when a solution is found (this could take up to 30 seconds), open the **Solution > a2** table. Point out that the data is there, but it's as difficult to interpret as it was in OPL. Tell them you're now going to create a chart to visualize the relationship of due dates to end times in the solution schedule.

Create the Due Dates Met chart

- In OPL, double-click on **sched_wood_for_odm_views.odmvw** to open its editor and right-click on the **Solution** folder. Select **Add Chart View** from the popup menu.
- Change the **View Name** of the chart to **Due Dates Met**.
- On the **Chart Tables** tab, click **Add**. On the popup, choose the **stands** table, choose **dueDate (Integer)** from the **Columns** dropdown list, and change its label from the default to **Due Date**. Then click **Finish**.
- Click the **Add** button again, choose the **a2** table, choose **e (Integer)** from the **Columns** dropdown list, and change its label to **End Date**. Click **Finish**.
- Switch to the **Chart Settings** tab and choose **Bar** from the **Rendering** dropdown list. Then choose **Clustered** from the **Type** dropdown list. Finally, choose **Outside Labels** from the **Annotation** dropdown list.
- Run the ODM application again and solve the problem. Open the **Solution** folder and display the new **Due Dates Met** chart.
- Point out that this chart is much easier to read than the **Solution > a2** table, and shows at a glance that all end dates in this schedule meet their due dates.
- You may want to end the demo by showing the students the non-user-friendly default mapping in place on the **Input** tables and the **Solution > a2** table. Remind them that this is easily changed, and that they did it themselves in their previous practice. Also tell them that they'll get a chance to create a chart themselves in one of the next lessons.

Extending ODM applications: custom visualizations

Learning objective
Learn how ODM charts can be designed in OPL and displayed in ODM

Key term
Chart Views

You are now going to examine, with your instructor, one of the important features of ODM, the ability to create customized visualizations of your information. This can help business users do “what if?” analysis.

The version of the Unit Commitment Problem application that you will use for this exercise has already been highly customized in OPL and using Java to create custom views and other extensions. Follow the steps below carefully, and **DO NOT** use the **ODM Application Generation Wizard** as you did in the previous topic. If you do, all of the customization in the application will be lost.

Practice

Work with ODM charts

In this step, you will examine the built-in charting capabilities of ODM, and how they can help the user visualize the data.

Go to the **Introduction to ODM** workshop and perform the **Create a chart view using the OPL IDE** step.

You can perform this lab using the HTML workshop, or by following the instructions in the workbook. The HTML workshop will give you direct access to OPL documentation pages that can help you with the lab.

Create a chart view by using the .odmvw editor

Objective

- *Create a new chart*

Action

- *Create a chart for the Average Cost KPI*

Reference

Adding a bi-indexed chart

The planner had a change of heart and would prefer to see the average cost information in the form of a chart. In this exercise, you'll create a chart to show the average cost per unit per period.

Create a new chart

1. *Import the* ***<TrainingDir>\OPL63.labs\ConfigViaOPLIDE\chart\work\ucp_chart_work*** *project into OPL. Do not check the* ***Copy projects into workspace*** *checkbox.*
2. *In the OPL Projects Navigator, expand ODM Application (ucp.mod) and double-click* ***ucp_views.odmvw****.*
3. *Expand the* ***Solution*** *and* ***KPI*** *directories. Right-click the KPI directory and choose* ***Add Chart View****. Change the name of the view to* ***`Average Cost per Unit Chart`.***
4. *Under the* ***Chart Tables*** *panel, choose the* ***Bi-indexed*** *radio button, because your data is bi-indexed. You can see this by looking at the data declaration in the OPL model:* ***`avgCostByUnitByPeriod[u in Units][t in Periods]`****, with* ***`u`*** *being the first index and* ***`t`*** *being the second index. Click* ***Add*** *and select the* ***Table*** *to be* ***avgCostByUnitByPeriod*** *,* ***Column*** *to be* ***value (Double)*** *and change the* ***Label*** *to* ***`Average Cost`****. Click* ***Finish****.*
5. *For the* ***X From*** *field select* ***Range Periods****, and enter 0 for the* ***Min*** *and 24 for the* ***Max*** *(to see only the first day's data), and for the* ***Chart for Each*** *field select* ***Units*** *to create a separate line for each unit.*
6. *Save all and launch the ODM application. Create a* ***New Default Scenario*** *renamed* ***`Chart Scenario`*** *and solve the scenario. Select the* ***Close this dialog box when solve completes*** *check-box.*
7. *When the solve has completed, return to the chart creation editor in OPL. Click the* ***Import Data from ODM*** *button to see how your chart will look in ODM (or look at it in ODM).*

8. *In OPL, select the* ***Chart Settings*** *tab and play with the different settings to see the impact on your chart display. For example, choose* ***Stacked*** *for the* ***Type*** *field.*
9. *Save all, close ODM and launch it again for your changes to take effect. Look at your* ***Average Cost per Unit Chart*** *again to make sure it looks like you intended. Compare with the view in the* **`chart\solution\ucp_chart_solution`** *directory if you need to make any changes.*
10. *Close ODM and delete any open projects from OPL before moving on to the next step (do not check the* ***Delete project contents on disk*** *box).*

Working with multiple scenarios in ODM Studio

Learning objective
Understand how to create and compare multiple scenarios in ODM.

Key terms
- scenarios
- reference scenario
- goals
- requirements
- relaxed requirements

Multiple scenarios for the same OPL model

The ability to create multiple scenarios in ODM Studio for the same OPL model is a power feature for “what-if” analysis.

Each scenario can be saved separately and modified separately in ODM Studio. Examples of possible modifications are:

- Changing the input data
- Activating, deactivating, constrain, and set priorities on goals
- Defining, activating and deactivating rules
- Setting relaxation priorities

Changes made to scenarios in ODM Studio do not affect the original OPL model.

Any new default scenarios will be created from the original OPL model. You can also copy any existing scenario, with any changes you might have already made to its parameters, and then make additional changes in the copy.

Comparing scenarios

Any two scenarios can be compared by doing the following:

- Choose one scenario as the reference scenario.
- Select the **Differences** check-box in the ODM **Legend** panel.
- Open any view in the non-reference scenario to see the differences between the two scenarios. The numbers corresponding to the reference scenario are shown in parentheses next to the numbers of the non-reference scenario.

Use Multi-Scenario Comparison Views to compare more than two scenarios.

You will now learn how to create and compare multiple scenarios in ODM Studio without modifying the original OPL model.

Practice

Create and compare multiple scenarios

In the UCP example, the “Spinning Reserve” refers to the maximum production capacity by all active generators minus the actual amount produced. In the following exercise you will create several scenarios with different Spinning Reserves expressed as percentages of the maximum production capacity, and then compare these scenarios.

You can either follow the instructions in the html workshop (which also include links to relevant documentation) or you can follow along in this book.

Create multiple scenarios and compare two scenarios

Objectives

- *Create multiple scenarios to cover different business use cases*
- *Compare two scenarios*

Actions

- *Import the project and launch ODM Studio*
- *Create scenarios*
- *Compare two scenarios*

References

tables
storing scenario data
scenario explorer
create a default scenario

Import the project and launch ODM Studio

1. *Import the* **`ucpBaseODMapp`** *project from the* **`<TrainingDir>\OPL63.labs\BaseODMapp\ucpBaseODMapp`** *directory. Do not check* ***Copy projects into workspace****.*
2. *Right-click* ***Run Configurations > Run ODM Application (default)*** *in the* ***OPL Projects Navigator*** *and choose* ***Run this*** *to launch ODM Studio.*

Create scenarios

You will now create four scenarios with different percentages for the Spinning Reserve, and compare two of these scenarios.

1. *If a* ***New Default Scenario*** *is present in the ODM application, leave it in place. If no scenarios are present, create one by choosing* ***File > New > New Default Scenario*** *from the main menu.*
2. *Create a second default scenario using the same procedure and rename it* ***Spinning Reserve 8%****.*
3. *Make sure that the* ***Spinning Reserve 8%*** *scenario is selected in* ***Scenarios Overview*** *window, and in* ***Scenario Explorer*** *click* ***Rules >Reserve Requirement*** *to display that window in the main window area.*
4. *Right-click* ***Reserve Requirement****, and choose* ***Add Rule >Spinning reserve should be at least <percent>% of load in each period*** *from the context menu.*

5. *In the rule editing area at the bottom of the window, click the* ***<percent>*** *field, type* ***8****, and press* **`<enter>`***. You have added a rule to this scenario that sets the reserve amount to 8%.*
6. *In* ***Scenarios Overview****, right-click* ***Spinning Reserve 8%*** *and choose* ***Duplicate Current Scenario*** *from the context menu. Rename this new scenario* ***Spinning Reserve 16%****.*
7. *On the* ***Reserve Requirement*** *window for this new scenario, change the reserve amount in the rule editing area to* ***16%****.*
8. *In* ***Scenarios Overview*** *again, right-click* ***Spinning Reserve 8%*** *and copy it a second time, renaming the new scenario* ***Spinning Reserve 24%****.*
9. *On the* ***Reserve Requirement*** *window for this new scenario, change the reserve amount in the rule editing area to* ***24%****.*
10. *Save all and solve each of the four scenarios.*

Compare two scenarios

You will now set a ***Reference Scenario*** *and compare it to one of the other scenarios.*

1. *In* ***Scenarios Overview****, right-click the* ***New Default Scenario*** *and choose* ***Use as reference*** *from the context menu. Note that the scenario name is now displayed in bold. This scenario will be used as a reference when comparing to the other three scenarios in the next steps.*
2. *Open the* ***Solution*** *folder in* ***Scenario Explorer*** *and click* ***Production Schedule*** *to display that window in the main window area. This is the solution schedule for the* ***New Default Scenario****.*
3. *Leaving the* ***Production Schedule*** *window open, select the* ***Spinning Reserve 24%*** *scenario in the* ***Scenarios Overview*** *window. You are now looking at the solution schedule for that scenario.*
4. *To compare it to the Reference Scenario, in the* ***Legend*** *panel at the lower left of the ODM application frame, check the* ***Differences*** *box. The differences between the solution found by the* ***Spinning Reserve 24%*** *scenario and the* ***New Default Scenario*** *become highlighted in gold. In the highlighted fields, note that the value displayed in parentheses is the value from the* ***Reference Scenario****, while the other value is for the currently-selected scenario.*
5. *Click on each of the other* ***Spinning Reserve*** *scenarios, and watch the* ***Production Schedule*** *window as the highlighted fields change.*
6. *If you want, open the* ***Solution > Production Pivot Table****, and do the same comparison. With the* ***Differences*** *option turned on in the* ***Legend*** *panel, you can easily see which fields have changed in the different scenarios, compared to the* ***Reference Scenario****. You can also change the* ***Reference Scenario****, and see how that affects what you see.*

You should now be able to see how, in an ODM application with multiple scenarios, you can generate comparisons between any two scenarios, using any of the existing scenarios as the Reference Scenario. However, using this procedure you are only able to compare two scenarios at the same time. What if you wanted to compare more than two scenarios at the same time?

Complete the **Compare multiple scenarios** step in the workshop to practice doing this.

Compare multiple scenarios

Objective

- *Use a Multi-Scenario View to compare three or more scenarios*

Actions

- *Add a Multi-Scenario View*
- *Compare multiple scenarios*

Reference

scenario comparison view

Add a Multi-Scenario View

You will now add a new Multi-Scenario Comparison View to the ODM application.

1. *Leaving the ODM application open, return to OPL and double-click the* ***ODM Application > ucp_views.odmvw*** *file.*
2. *In the left pane of the editor, right-click the* ***Unit Commitment Demo*** *item at the top and choose* ***New Group*** *from the context menu. In the popup window, name the new group* ***MSV*** *and click* ***Finish****.*
3. *Next, right-click the new* ***MSV*** *group and choose* ***Add Multi-Scenario Comparison View*** *from the context menu. A new view is created, and an editor for the view appears in the right pane of the editor.*
4. *In the right pane of the editor, change the* ***View Name*** *field to* ***Multi-Scenario Comparison View****.*
5. *Save and launch the* ***ucpBaseODMapp*** *ODM application again. You will be prompted to save your changes, and the ODM application will automatically be shut down and relaunched, with the new view in place.*

Compare multiple scenarios

Now compare all four scenarios in the new ***Multi-Scenario Comparison View*** *window.*

1. *In ODM Studio, select the* ***MSV > Multi-Scenario Comparison View*** *window in any of the scenarios. In the next step you will add the data for all the scenarios.*
2. *Click the* ***Configure Table*** *button in the top far right side of the window. (It is the topmost of the three buttons; you can mouseover the icons to see their tooltips to locate the right one.)*
3. *In the* ***Configure Table*** *popup window that appears, check the boxes for each of the four scenarios and click* ***OK****. The window is displayed, now containing all four scenarios. If necessary, drag the top pane of the window down so that you can see all four in the table area at the top.*
4. *Note that this view provides a high-level overview of each of the four scenarios and their solutions. In the table area, you can see the* ***Scenario*** *name, followed by the two goals of the UCP application —* ***Fuel Cost*** *and* ***Start Up Cost****. The values that you see in each column are the same that you would see in the* ***Goals*** *window for each of the scenarios. Next is the* ***Number Of Periods*** *for each scenario.*
5. *The next two columns show the solution status for the scenarios. A check in the* ***Has Result*** *column indicates that this scenario ran to completion and found a solution. A check in the* ***Is Feasible*** *column indicates that it is a feasible solution. Note that one of the scenarios,* ***Spinning Reserve 24%****, is displayed in red. This is because that scenario's solution required relaxations of at least one of the requirements.*
6. *The* ***Gap*** *column displays the relative difference between the integer solution found and the proven best possible objective solution value.*
7. *The lower part of the* ***Multi-Scenario Comparison View*** *window displays the goal results in chart format, by default a Bar chart. To change the configuration of this chart, click the* ***Configure Chart*** *button*

in the top right of the window. (It is the middle of the three buttons.) On the resulting popup window, you can choose between normal and 3D Bar charts, and normal and 3D Polyline charts.

8. *To remove one or more scenarios from the view, highlight the one you want to remove in the chart area and click the* ***Remove Selected Rows*** *button in the top right of the window. (It is the bottom of the three buttons.)*

You've now seen how you can compare multiple scenarios using a Multi-Scenario Comparison View.

ODM Requirements

Learning objective
Understand how requirements are generated from relaxed OPL constraints and how they can be modified in ODM.

Key terms
- requirements
- soft constraint
- requirement view

An OPL model contains constraints. These constraints can be mapped to ODM **requirements** to be candidates for relaxation. In Operations Research terms, requirements are mapped to **soft constraints.**

OPL constraints can be simple constraints, or indexed constraints in a **`forall`** declaration. To be mapped to requirements, these constraints need to be labeled in the OPL model.

Only requirements that have actually been relaxed by the **`solveAnyway`** algorithm are shown in the **Requirements** view of ODM Studio.

Each relaxed requirement is displayed in ODM Studio in a tree structure. The top level consists of a folder that has the same name as the labeled constraint in the OPL model. The next level of the tree structure shows the **grouping** of this requirement.

Each requirement is assigned a priority by the *priority propagation scheme* assigned to the requirement when the ODM application is generated. The ODM Application Generation Wizard assigns a default priority of **Parent** to each requirement in a group. This means that each of the child levels of the requirement's tree structure inherits its priority from its parent in the tree, and by default, the top level of each requirement group is assigned the priority **Medium.**

Priorities can be changed in order to try and find better solutions to the problem. There are two ways to do this:

- In the ODM Studio **Requirements** view.

  Priority changes made in this way are temporary, and are not automatically saved when you exit from the ODM application; you need to explicitly save.

  Because requirements only appear in this view when the corresponding constraints have been relaxed during a solve, you can only use this method of changing priorities if the corresponding constraint has been relaxed.

- In the IDE **Requirements editor.** These changes are automatically saved, and become part of the generated ODM application.

Continue with the next step of the workshop to practice modifying requirements.

Practice

Modify requirements

Objective

- *Learn how an end user can work with constraint relaxation from within the ODM application*

Action

- *Change requirements, relaxations and priorities*

Reference

Changing the priority of a requirement in ODM Studio

Change requirements, relaxations and priorities

*In ODM Studio, go to the **Scenario Explorer** and select **Analysis > Requirements**. As before, this window is blank, because no constraints have been violated. Requirements are only visible in ODM when a constraint was relaxed in order to find a feasible solution. You are now going to change the demand data to demonstrate this principle.*

1. *In the **Scenario Explorer**, go to **Input Data > Load** and change the Load data for the first 7 periods to 3000.*

   *Note how easy it is for an end-user to change data. Changes made in this way are not written to the OPL **`ucp.dat`** file. The data is only changed to the particular scenario in ODM.*

2. *Solve the model again. You will be asked to save the scenario. Click **Save and Continue** to proceed with the solve. While it is running, notice that the **Solve Progress** window displays the message (at the bottom), "Searching for a solution enforcing requirements above Medium." This means that there is a conflict or other infeasibility somewhere in the model, and that constraints with a priority (as set in OPL or changed locally in ODM) at or below "Medium" are candidates for relaxation.*
3. *When the solve finishes, close the **Solve Progress** window and go to the **Scenario Explorer**. See that the word **Requirements** is now shown in red. The **Legend** window tells you that this means there are relaxed requirements.*
4. *Return to the **Requirements** window and note that the **meet_demand** requirement (and also some others) has been relaxed. In other words, the demand was not always met. Expand this item to see for which particular generators and time periods the constraint was relaxed for.*

   *The message in the **Relaxation** column shows the production values found in the relaxed solution, as well as the original demand target.*

5. *Now look at the **Priority** column. Each of the levels of the tree structure indicates that it has a priority of **Medium**.*

   *This is because the default priority propagation scheme assigned to the requirement by the ODM Application Generation Wizard is **Parent.** This means that each of the child levels of the requirement's tree structure inherits its priority from its parent in the tree. The parent requirement in this case was assigned the default value of **Medium**, so all of its children inherit that priority.*

*Click the word **Medium** in the **Priority** column of the **for 1** requirement. Select **Mandatory** from the drop-down list and note the changes in the window:*

- *The item is highlighted in color.*
- *The **Priority Modified** checkbox is checked.*
- *The requirement is shown as modified in the **Summary of changes to the model** pane below the list.*

6. *Save and solve the model again. Close the **Solve Progress** window when done, and return to the **Requirements** window.*

   *You can see that **for 1** no longer appears in the list of relaxed requirements. Its priority was changed, and ODM has had to relax many other requirements in order to meet the mandatory load requirement in the first period. Examine the other relaxed requirements.*

   If a constraint in the OPL model has not been relaxed, it will not appear as a requirement in ODM.

7. *Expand the **oper_max_generation** requirement, and expand **COAL_1**. Note that there are two periods grouped under it. The default grouping for this requirement, then, is first by unit (u) and then by period (t) from the OPL model. This can be changed when configuring the application.*
8. *Click the word **Medium** in the **Priority** column of the **oper_max_generation** requirement. Select **Low** from the drop-down list. Note that all the branches and leaves of this requirement have changed their priority to Low. This is because their default priority setting is **Parent**.*

Requirements are constraints that could be relaxed. If a constraint is not declared as a requirement, ODM will not consider that constraint for relaxation when solving the model. Any hard constraints, for example that a unit must be turned on for it to be in use, should not be declared as requirements.

Copy to Microsoft Office

As a last "stop" on this tour, we will show you one of the built-in features of ODM that could be very useful to your users. Often users want to be able to export the solution data into a spreadsheet for further analysis. With ODM, this can be done in one click.

The following exercise uses the **`<TrainingDir>\OPL63.labs\BaseODMapp\ucpBaseODMapp`** example. If it is not already open in OPL, open it and launch the ODM application.

Export your schedule to Excel or other Microsoft Office applications

1. In any scenario that contains a solution (that is, one that has been solved), open **Solution > Production Schedule** to display the final schedule for the scenario.
2. Remove any filters that may have been created, so that the entire table is displayed.
3. Click in the window, then press **`<ctl>A`** to select the entire table.
4. Right-click the selected area and select **Copy** from the context menu (or press **`<ctl>C`**). The contents of the table have been copied to the Windows Clipboard.
5. Open Microsoft® Excel and, in the blank spreadsheet that appears, press **`<ctl>-V`** to paste the data into Excel. The data is pasted into Excel, including the column headings.

   If you want to cut and paste data without the column headings, right-click in the table before copying it, and deselect **Includes Headers In Copy**. This is a toggle switch, and is either on or off.

   You can also select a smaller group of cells in the table, and only the selected cells will be cut and pasted, with or without the column headings.

You can use this method to cut and paste from most ODM views, including pivot table views. The destination application can be Excel, PowerPoint, Word, or any of the Microsoft Office applications.

Summary

Review

In this lesson, you learned about the IBM ILOG Optimization Suite and integration between OPL and ODM:

- OPL models can be used to quickly generate an ODM end-user application.
- You can design graphs and charts to visualize both input data and solution results, and have them displayed in ODM.
- The user can create multiple scenarios and change goals, constraints and bounds, all without affecting the original OPL model.
- Scenarios can be compared, saved for later use, and/or exported to Excel spreadsheets.

If you want to see additional features of ODM, we suggest that you complete the **ODM Walkthrough**, contained in the ODM documentation. That document takes you on a more in-depth tour of how to manage different scenarios in ODM and shows you more ways that you can work with them to come up with the best solution. It also demonstrates several additional built-in features of ODM, and a few custom views that have been created for this example using Java.

Lesson 15: Flow Control with IBM ILOG Script

Instructor note
This is an optional lesson. This lesson should last about 2 hours, including the practices. This lesson assumes that the students have already learned how to read and write from a database. The lab (Staffing) uses an MS Access database (the files are given with the labs, but you need to have Access installed on the students' computers as well as your own).

IBM® ILOG® Script is embedded in OPL to enable pre- and postprocessing, as well as flow control. In an earlier lesson you learned how to use IBM ILOG Script for pre- and postprocessing. In this lesson, you'll learn how to use IBM ILOG Script for flow control.

At the end of this lesson you will be able to:

- Understand the difference between JavaScript™ and the IBM ILOG Script extensions for OPL
- Implement some of the flow control functionality available in OPL, specifically:
    - Accessing and modifying model and data elements
    - Modifying the CPLEX® matrix incrementally
    - Looping controls
    - Warm start
    - Integer relaxation
    - Postprocessing
    - Debugging

This lesson includes a practice and an example that shows a column generation implementation using both CPLEX and CP Optimizer.

IBM ILOG Script extensions for OPL

Learning objective

Understand what the IBM® ILOG® Script extensions for OPL are.

Key terms
- OPL extension classes
- JavaScript

What is IBM ILOG Script?

- IBM ILOG Script is an implementation of the ECMA-262 standard (also known as ECMAScript or JavaScript™)
- This implementation also includes extension classes for OPL
- The extensions for OPL enable script blocks to access and manipulate OPL elements
- The same extension classes are used in the OPL APIs – IBM ILOG Script offers a subset (some methods and concepts are only available in the APIs).

The IBM ILOG Script extension classes for OPL are essential for flow control. In this topic you'll learn what the extension classes are conceptually, and how to find more information on these classes.

JavaScript vs. OPL extension classes

In the earlier lesson on basic script functionality, you learned how to use IBM ILOG Script to perform some basic tasks such as pre- and postprocessing.

Some of the keywords required to perform these tasks correspond to JavaScript™, for example **`for`** and **`writeln`**.

However, accessing the OPL elements requires using the IBM ILOG Script extension classes. These are classes that extend JavaScript to meet the needs of OPL users.

All the IBM ILOG Script extension classes for OPL start with **`Ilo`**, for example **`IloOplModel`** or **`IloCplex`**, and are therefore easily recognizable.

Note that even though extension classes are used to access model elements in an OPL script block, they are not always explicitly visible in the code, for example in the following **`execute`** block that accesses the tuple set **`Workers`**, as well as the tuple elements **`salary`** and **`raise`**:

```
execute {
   for (var w in Workers) {
      w.salary = w.salary * w.raise;
   }
}
```

In other cases, such as calling a model in a **`main`** block, the call to the extension class is visible, for example:

```
main {
 var source = new IloOplModelSource("basicmodel.mod");
 var def = new IloOplModelDefinition(source);
 ...
}
```

In the remainder of this lesson, you'll see how the IBM ILOG Script extension classes for OPL are used to implement flow control using IBM ILOG Script **`main`** blocks. While you'll learn more about some of the extension classes and their methods in this lesson, please refer to the **IBM ILOG Script Reference Manual** in the documentation for a complete list and explanation.

Instructor note
The class library documentation previously in this topic has been removed to permit you to show whichever classes you might want to explore with your students, as a function of needs and student ability. While several of the extension classes will be covered in the remainder of this lesson, feel free at this point to put the documentation up on the screen and explore individual classes if you think this is useful for your student group. This documentation will always be the latest and thus up to date.

Flow control and the main block

Learning objective
Understand what is meant by flow control and how to implement it using a **`main`** block.

Key terms
- flow control
- main block

What is flow control?

Flow control enables you to control how models are instantiated and solved, for example to:

- solve several models with different data in sequence or iteratively
- run multiple “solves” on the same base model, modifying data and/or constraints after each solve
- decompose a model into smaller more manageable models, and solve these to arrive at a solution to the original model (model decomposition)

Some examples where you may want to use IBM® ILOG® Script for flow control are:

- Implementing column generation (a mathematical programming algorithm)
- Decomposing a supply chain model into a planning model and a scheduling model and solving these in sequence
- Using one model to create a “warm start” for another model

Flow control can also be implemented using the available OPL APIs, for example for implementations in C++ or Java. The methods used in such implementations are similar to the ones you'll learn about in this lesson.

Practice

Discussion

Can you think of some other examples where flow control can be useful? Discuss with the instructor and the class.

Instructor note
Some topics you can discuss here depending on the level of your audience:

- Lagrangean decomposion/relaxation.
- Aggregating a set of smaller time periods for a long-term planning problem into a set of larger time periods (e.g. aggregating 52 weekly periods into 12 periods of 1 week each, 2 periods of 4 weeks each, and 2 periods of 16 weeks each), and finding a solution to this aggregate problem. You can then disaggregate the time periods into the original 52 periods and use parts of the solution to the aggregate problem to help find a solution for each of the original time periods.
- Decomposing a problem that contains both tactical and operational decisions into an upper level tactical planning problem and a lower level operational planning problem.

The main block

To implement flow control, you must add a **main** block to your **.mod** file using this general syntax:

```
main {
...
}
```

Each **.mod** file can contain at most one **main** block, and the **main** block will be executed first, regardless of where it is placed in the **.mod** file.

What is a model instance?

A model instance is a combination of a model and data. Two model instances in a **main** block can use the same model definition (that is, the same **.mod** file), or different model definitions from other **.mod** files.

A main block using CPLEX

The following example shows some simple and often used content of a **main** block when solving a model with CPLEX:

```
main {
 thisOplModel.generate();
 if (cplex.solve()) {
  var obj=cplex.getObjValue();
 }
}
```

- **thisOplModel** is an IBM ILOG Script variable available by default referring to the current model instance, that is, the model definition that contains the **main** block currently executed together with the associated **.dat** files (if they exist).
- **generate()** is a method used to generate the model instance.

- **`cplex`** is an IBM ILOG Script variable available by default, that refers to the CPLEX instance.
- **`solve()`** calls CPLEX to solve the model.
- **`getObjValue()`** is a method to access the value of the objective function.

A main block using CP Optimizer

The following example shows some simple and often used content of a **`main`** block when solving a model with CP Optimizer:

```
main {
 thisOplModel.generate();
 if (cp.solve()) {
  var obj=cp.getObjValue();
 }
}
```

Many of the methods available when using CP Optimizer are the same as for CPLEX. The only difference in the example above is that **`cp`** is an IBM ILOG Script variable that refers to the CP Optimizer instance.

If the **`main`** block is in a model file starting with **`using CP;`**, the **`cp`** variable is available by default, but you'll have to declare the **`cplex`** variable explicitly if you want to call **`cplex`** in that same **`main`** block:

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

Conversely, if you do not state **`using CP;`** in the model file containing the **`main`** block, you will have to declare the **`cp`** variable explicitly if you want to solve using CP Optimizer.

Templates to construct a main block

You can work with models and data in a **`main`** block by

- Calling a project and creating a run configuration, or
- Calling a model and data

You can use the following two templates as starting points for writing flow control script. These will come in handy for the exercise later in this lesson.

Flow control script template calling a project:

```
main {
 var proj = new IloOplProject("../../../../../opl/mulprod");
 var rc = proj.makeRunConfiguration();
 rc.oplModel.generate();
 if (rc.cplex.solve()) {
  writeln("OBJ = ", rc.cplex.getObjValue());
 }
 else {
  writeln("No solution");
 }
 rc.end();
 proj.end();
}
```

This main block first creates a variable **`proj`** that calls the OPL project **`mulprod.prj`**. It then uses the **`makeRunConfiguration()`** method to create a variable, **`rc`**, for the run configuration. Because no particular run configuration name is given as an argument to the method **`makeRunConfiguration`**, it will use the default run configuration as configured in the project, but one can also specify the name of a particular run configuration. From the **rc** variable, you can then access the OPL model instance and the **cplex** instance to call other methods such as **generate()**, **solve()** and **getObjValue()**.

Flow control script template calling a model and data:

```
main {
 var source = new
IloOplModelSource("../../../../../opl/mulprod/mulprod.mod");
 var def = new IloOplModelDefinition(source);
 var opl = new IloOplModel(def,cplex);
 var data = new
IloOplDataSource("../../../../../opl/mulprod/mulprod.dat");
 opl.addDataSource(data);
 opl.generate();
 if (cplex.solve()) {
  writeln("OBJ = ", cplex.getObjValue());
 }
 else {
  writeln("No solution");
 }


 opl.end();
 data.end();
 def.end();
 source.end();
}
```

This main block first creates a variable **source** that calls the OPL model. It then creates the **def** variable for the model definition using **source**, and the **opl** variable for the model instance. Next, it creates the **data** variable for the data source (**mulprod.dat**) and adds this data source to the **opl** model instance.

The **opl** model instance is then used to generate the model, and the **cplex** instance is used to call the **solve()** and **getObjValue()** methods.

While these two templates result in similar behavior (generating and solving a model), you should use the first if you'd like to add a settings file in the script, because you can only add a settings file using a run configuration.

Ending objects in a main block

In the given templates, the **end()** method is called to end objects. It is good practice to use this method to systematically terminate objects that are no longer necessary.

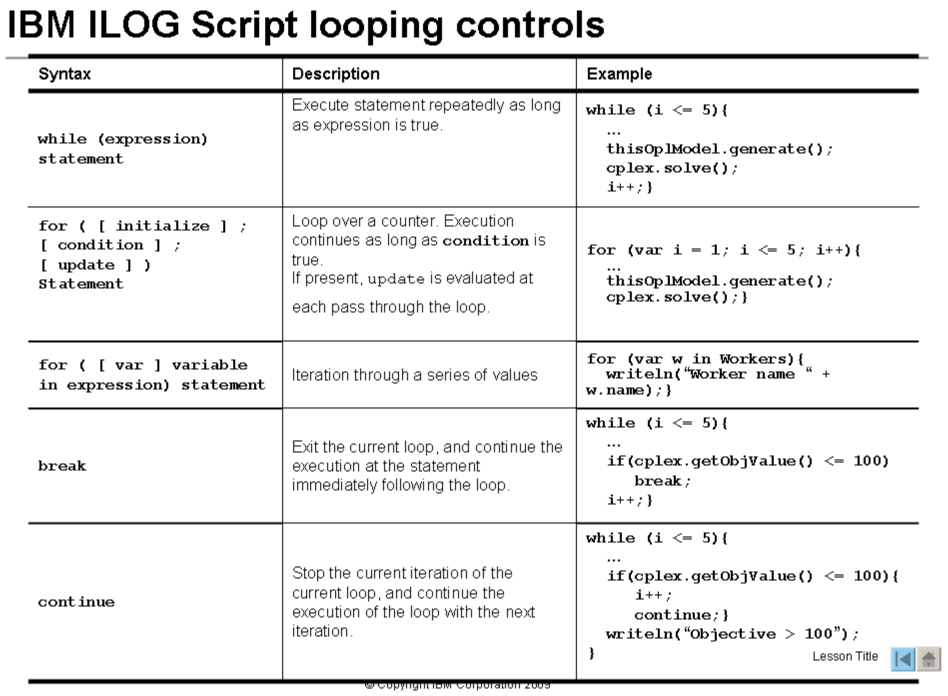
The **end()** method is disabled by default in the OPL IDE. You can enable it by setting **mainEndEnabled** to true using the following script statement:

```
thisOplModel.settings.mainEndEnabled = true;
```

We recommend that you use caution applying this setting. Faulty memory management by the user, such as attempting to use an object after it has been deleted may result in crashes.

Looping controls

The following table summarizes the looping controls available in IBM ILOG Script:



IBM ILOG Script looping controls

Syntax	Description	Example

`while (expression)`
`statement`
	Execute statement repeatedly as long as expression is true.	`while (i <= 5){`
		`…`
		`thisOplModel.generate();`
		`cplex.solve();`
		`i++;}`

`for ( [ initialize ] ;`
`[ condition ] ;`
`[ update ] )`
`Statement`
	Loop over a counter. Execution continues as long as **`condition`** is true.
	If present, `update` is evaluated at
	each pass through the loop.
		`for (var i = 1; i <= 5; i++){`
		`…`
		`thisOplModel.generate();`
		`cplex.solve();}`

`for ( [ var ] variable`
`in expression) statement`
	Iteration through a series of values	`for (var w in Workers){`
		`writeln("Worker name " +`
		`w.name);}`

`break`	Exit the current loop, and continue the execution at the statement immediately following the loop.	`while (i <= 5){`
		`…`
		`if(cplex.getObjValue() <= 100)`
		`break;`
		`i++;}`

`continue`	Stop the current iteration of the current loop, and continue the execution of the loop with the next iteration.	`while (i <= 5){`
		`…`
		`if(cplex.getObjValue() <= 100){`
		`i++;`
		`continue;}`
		`writeln("Objective > 100");`
		`}`

Lesson Title

© Copyright IBM Corporation 2009

Note that if the condition for the **`for`** loop is omitted, it is taken to be true, producing an infinite loop. If this condition gives a non-Boolean value, this value is converted to a Boolean value.

There is no range syntax in IBM ILOG Script loops. For example,

`for(var i in 1..n)`

will yield an empty loop.

For more detail see **Language > Language Reference Manual > IBM ILOG Script for OPL > Language structure > Statements** in the OPL documentation.

Model and data access

Learning objective
Understand how to access model and data elements in a **`main`** block, and how to modify the model directly in CPLEX® or CP Optimizer without affecting the original OPL model. Learn how to create an integer relaxation and implement a warm start.

Key terms
- Data access
- Model access
- Integer relaxation
- Warm start

Data access

Data for a particular model instance can be accessed and/or changed in a **`main`** block, for example to run an iterative solve on the same model, where data is changed with each iteration.

To change data during flow control before regenerating the model, you must get the data elements from the **`IloOplModel`** instance, for example:

```
main {
 ...
 var data = thisOplModel.dataElements;
 ...
}
```

When calling the method **`dataElements`** on an **`IloOplModel`** instance, you obtain a container of all the data elements of this model. The original data source (e.g. the **`.dat`** file) remains unchanged – you are only changing data associated with a particular model instance.

Data access example

The following example shows how to solve a model, edit a data element, and resolve the same model with the changed data. It assumes the model data includes an array called **`Capacity`**, indexed over different products. In the example below the capacity is changed for "**`flour`**" before generating and solving the model a second time.

```
main {
// generate and solve the initial model
 var initialModel = thisOplModel;
 initialModel.generate();
 if( cplex.solve() )
  writeln("The objective with initial capacity ", flourCapacity, "
 is ", cplex.getObjValue());

// access and edit the flour capacity
 var flourCapacity = initialModel.Capacity["flour"];
 flourCapacity = flourCapacity + 10;



// create a new data instance with the new capacity
 var data = initialModel.dataElements;
 data.Capacity["flour"] = flourCapacity;

// create a new model instance using the same model defintion, with
 the updated data
 var def = initialModel.modelDefinition;
 var updatedModel = new IloOplModel(def,cplex);
 updatedModel.addDataSource(data);

// generate and solve the updated model
 updatedModel.generate();
 if( cplex.solve() )
```

```
  writeln("The objective with updated capacity ", flourCapacity, "
 is ", cplex.getObjValue());
}
```

There is no tuple syntax available in **main** blocks.

Instead, use the **find()** and **get()** methods to control tuple objects

Example:

Let's say, in the preceding example, that the **Capacity** is an array indexed over a tuple set **productsOnMachines** with tuple components **product** and **machine**

In that case, instead of using tuple syntax such as:

```
Capacity[<product>,<machine>]
```

you need to use:

```
Capacity[productsOnMachines.get(product,machine)]
```

Only data external to the model can be modified using IBM® ILOG® Script (e.g. data declared in the **.mod** file, but initialized in a **.dat** file).

Data elements that are initialized within the model file cannot be modified using IBM ILOG Script.

Scalar data, whether in the **.mod** file or the **.dat** file, cannot be modified via scripting. Scalar data is the simplest form of data (for example, **int i = 4** or **float demand = 10**), where the data item does not have a more complex structure such as a set, array or tuple.

While the method you just saw for changing data is very useful, the model needs to be generated each time after a modification.

Sometimes, when solving iteratively with a large number of iterations, generating the new iteration takes a long time compared to solving it.

In this case, you may prefer to have a direct interaction with the generated optimization model to be able to work incrementally on the result of the previous iteration.

Modifying the CPLEX/CP Optimizer model directly

IBM ILOG Script allows you to modify the CPLEX®/CP Optimizer model directly to:

- Change the bounds on a constraint
- Change the bounds on a variable
- Change the variable coefficient in an objective or constraint

When you use this technique, the CPLEX/CP Optimizer model is directly modified but the OPL model is not.

Therefore, the solution given by CPLEX/CP Optimizer corresponds to the modified model (in memory), but no longer to the original OPL model that you see in the IDE.

The advantage is that the CPLEX/CP Optimizer model is directly modified (not rebuilt from scratch) and any new search can take advantage of the previous ones.

The following example shows how to modify a constraint upper bound directly in the CPLEX/CP Optimizer model. Instead of writing

```
        data.Capacity["flour"] = flourCapacity;
```

to edit the **`Capacity`** data element, you can edit the upper bound on the capacity constraint directly, where we assume in this case that **`ctCapacity`** is the name (or label) of this constraint:

```
thisOplModel.ctCapacity["flour"].UB = flourCapacity;
```

You can use similar syntax to modify variable bounds, and the **`setCoef`** method can be used to change variable coefficients.

Using this technique, the changes are only in the CPLEX/CP Optimizer model, and the OPL model is not affected. On the other hand, the change is taken into account incrementally by the CPLEX/CP Optimizer engine.

See **IBM ILOG Script for OPL > Tutorial: Flow control and multiple searches >Modifying the CPLEX matrix incrementally** in the **Language User's Manual** of the documentation for more information.

Integer relaxation

IBM ILOG Script provides a simple way to perform integer relaxation, that is, relaxing the integer requirement on decision variables to convert a MIP to an LP. Simply call the following method:

```
IloOplModel.convertAllIntVars
```

To undo a relaxation, call:

```
IloOplModel.unconvertAllIntVars
```

Accessing reduced costs, duals and slacks

IBM ILOG Script also allows you to access the following information for CPLEX models:

- Reduced costs for variables (**`variableName.reducedCost`**)
- Dual values for constraints (**`constraintName.dual`**)
- Slack values for constraints (**`constraintName.slack`**)

This information can be used for sensitivity analysis.

For more information, see **Language > Language User's Manual > The application areas > Applications of linear and integer programming > Linear programming** in the OPL documentation.

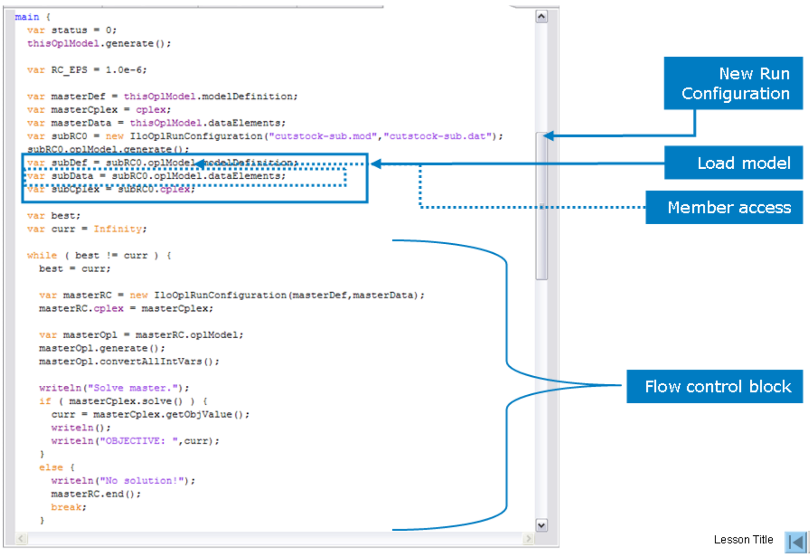
Practice

Example: Data access and flow control using a while loop

Launch the OPL IDE and import the project**`<OPLhome>\examples\opl\cutstock`** supplied with the product, where **`<OPLhome>`** represents the top level directory where OPL is installed.

Look at the **`.mod`** file called **`cutstock_int_main`**. The **`main`** block starts at line 30. The following diagram highlights certain elements of data access and flow control. A new run configuration is generated, starting at line 39.

Data access and flow control



```
main {
  var status = 0;
  thisOplModel.generate();

  var RC_EPS = 1.0e-6;

  var masterDef = thisOplModel.modelDefinition;
  var masterCplex = cplex;
  var masterData = thisOplModel.dataElements;
  var subRC0 = new IloOplRunConfiguration("cutstock-sub.mod","cutstock-sub.dat");
  subRC0.oplModel.generate();
  var subDef = subRC0.oplModel.modelDefinition;
  var subData = subRC0.oplModel.dataElements;
  var subCplex = subRC0.cplex;

  var best;
  var curr = Infinity;

  while ( best != curr ) {
    best = curr;

    var masterRC = new IloOplRunConfiguration(masterDef,masterData);
    masterRC.cplex = masterCplex;

    var masterOpl = masterRC.oplModel;
    masterOpl.generate();
    masterOpl.convertAllIntVars();

    writeln("Solve master.");
    if ( masterCplex.solve() ) {
      curr = masterCplex.getObjValue();
      writeln();
      writeln("OBJECTIVE: ",curr);
    }
    else {
      writeln("No solution!");
      masterRC.end();
      break;
    }
```

Lesson Title

© Copyright IBM Corporation 2009

In this example you saw how a **`while`** loop can be used for flow control. In this particular case, the **`while`** loop is part of a column generation implementation – you'll learn more about column generation later in this lesson.

Instructor note
In the course of this practice, you may want to demonstrate how you can use the problem browser to view the IBM ILOG Script call stack.

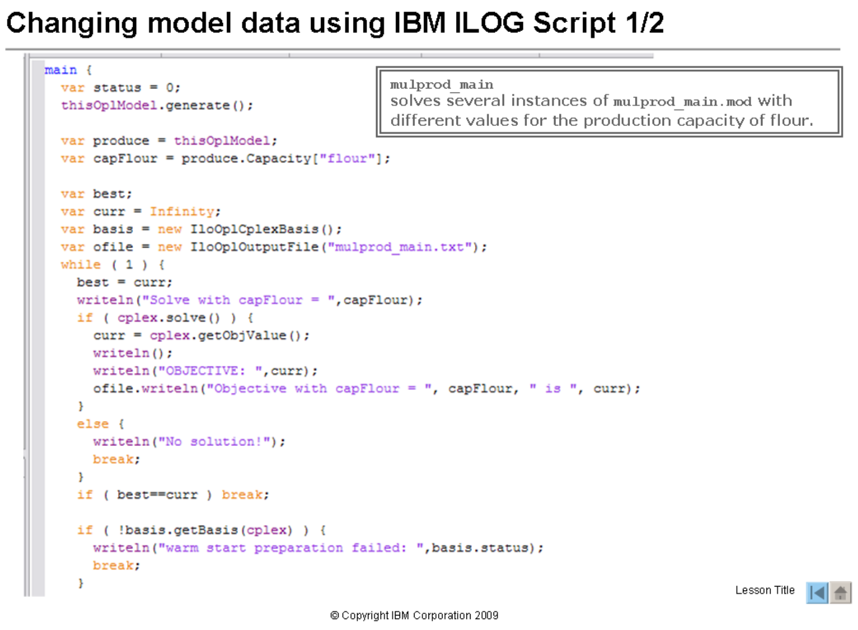
Practice

Example: Accessing and changing data elements

Import the project **`<OPLhome>\examples\opl\mulprod`** found in the product example files directory.

This multiperiod production planning example is a generalization of the single-period **Pasta Production** workshop you did earlier. The multiperiod version considers the demand for the products over several periods and allows the company to produce more than the demand in a given period.

There is an inventory cost associated with storing the additional production. Look at the **`.mod`** file called **`mulprod_main`**.



Changing model data using IBM ILOG Script 1/2

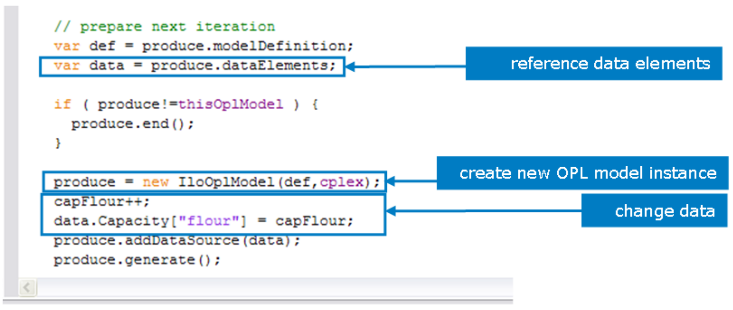
```
main {
  var status = 0;
  thisOplModel.generate();

  var produce = thisOplModel;
  var capFlour = produce.Capacity["flour"];

  var best;
  var curr = Infinity;
  var basis = new IloOplCplexBasis();
  var ofile = new IloOplOutputFile("mulprod_main.txt");
  while ( 1 ) {
    best = curr;
    writeln("Solve with capFlour = ",capFlour);
    if ( cplex.solve() ) {
      curr = cplex.getObjValue();
      writeln();
      writeln("OBJECTIVE: ",curr);
      ofile.writeln("Objective with capFlour = ", capFlour, " is ", curr);
    }
    else {
      writeln("No solution!");
      break;
    }
    if ( best==curr ) break;

    if ( !basis.getBasis(cplex) ) {
      writeln("warm start preparation failed: ",basis.status);
      break;
    }
```

`mulprod_main`
solves several instances of `mulprod_main.mod` with different values for the production capacity of flour.

Lesson Title

© Copyright IBM Corporation 2009

Changing model data using IBM ILOG Script 2/2



```
// prepare next iteration
var def = produce.modelDefinition;
var data = produce.dataElements;

if ( produce!=thisOplModel ) {
  produce.end();
}

produce = new IloOplModel(def,cplex);
capFlour++;
data.Capacity["flour"] = capFlour;
produce.addDataSource(data);
produce.generate();
```

Only data external to the model can be modified. Data elements that are defined inline within the model file cannot be modified.

Lesson Title

The **`main`** block starts at line 50 and initiates 2 local script variables, **`produce`** and **`capFlour`.**

Starting at line 60 you can see an example of using the **`while`** keyword to iterate through data and solve multiple models (see diagram above).

The program changes the quantity of flour available until the solution can't be improved any more.

Practice

Example: Changing data directly in the CPLEX matrix

In the **`mulprod`** project, open the file **`<OPLhome>\examples\opl\mulprod_change_main.mod.`**

This file shows you how the example can be modified to change the optimization model directly (i.e. without generating a new CPLEX model). In particular, line 76 changes the bound of a constraint:

```
for(var t in thisOplModel.Periods)
  thisOplModel.ctCapacity["flour"][t].UB = capFlour;
}
```

Postprocessing and debugging

Learning objective
Understand how postprocessing is handled when called inside a **`main`** block. Learn how to debug a **`main`** block.

Key terms
- postprocessing
- debugging

Postprocessing from a main block

When a model is executed in a **`main`** block, the postprocessing part is not executed by default. To execute a postprocessing block, call the **`postProcess()`** method:

```
main{
 ...
 thisOplModel.postProcess();
 ...
}
```

Debugging a flow control script

The debug features in the IDE are:

- breakpoint management
- call stack display
- variable object examination
- stepping

A simple debugging scenario is to place a breakpoint in a script, execute the script by means of the Debug button, examine the call stack, and then interactively execute statements using the Step Over button.

Practice

Debugging mulprod_main

Go to **OPL IDE > IDE Tutorials > Using IBM ILOG Script for OPL > The multiperiod production planning example** in the OPL documentation, and perform the last step entitled **Debugging a flow control script**.

Lab – The Staffing Problem

Learning objective

Practice using the IBM® ILOG® Script functionality learned thus far.

Key terms

- integer relaxation
- model access

Practice

The Staffing Problem

In this lab, you'll get to practice using the principles of flow control learned thus far.

In the first exercise, you'll implement a heuristic solution process where the solution to a relaxed model is used to create a warm start for the original model.

In the second exercise, you'll access model elements to iteratively improve a solution by editing constraint bounds directly in the CPLEX® matrix.

Go to the **Staffing Problem** workshop and perform the following steps, in order:

- **LP relaxation script**
- **Model access script**

LP relaxation script

Objective

- *Use IBM ILOG Script to write a simple solution heuristic*

Actions

- *A heuristic solution approach*
- *Steps to implement the heuristic*

Reference

IloOplModel

Exercise folder

<trainingDir>\OPL63.labs\Staffing\Scripted_models\LP_Relaxation\work

A heuristic solution approach

For some complex MIP models, it may take prohibitively long to find even a first feasible solution, not even to mention an optimal solution. In such cases, it could be very useful to use a heuristic approach in order to find a partial MIP solution, and then to feed that solution as an advanced starting point to the original model. CPLEX's default settings are such that if an advanced start is available in memory, it will be used in a subsequent solve and you therefore do not need to do anything special to ensure that this advanced information will be used, except for solving the models in sequence using IBM® ILOG® Script (or an API).

In this exercise, you will practice implementing such a heuristic for the staffing problem. The steps of the heuristic you'll be implementing are as follows:

1. *Solve the LP relaxation of the original MIP model.*
2. *Fix all Boolean decision variables that have a relaxed value above 0.6 to 1.*
3. *Solve the MIP with the fixed subset of Boolean decision variables. Note that the model is still guaranteed to be feasible, because we've only forced a subset of workers to be hired and we haven't forced any exclusions. The MIP now has a reduced set of decision variables, because a subset of them has been fixed. This would be especially useful for larger models where the size of the MIP could be significantly reduced. The solution to this model is not necessarily optimal, but you're not interested in*

optimality at this point seeing that you're only using the solution as a starting point for the original model.

4. *Use the solution from Step 3 as an advanced starting point for solving the original MIP.*

Steps to implement the heuristic

1. *Import the* **`staffing2Work`** *project in the* **`<trainingDir>\OPL63.labs\Staffing\Scripted_models\IP_Relaxation\work`** *directory. Leave the* ***Copy projects into workspace*** *box unchecked. You'll write your heuristic inside the* **`main`** *block. Some script statements to print intermediate values to the* ***Scripting log*** *output tab are already included.*
2. *Generate the original model using the* **`thisOplModel.generate()`** *command, relax the model using the* **`thisOplModel.convertAllIntVars()`** *command, and solve the relaxation using the* **`cplex.solve()`** *command*
3. *Test each Boolean decision variable in the array* **`hireWorker[namesOfWorkers]`** *for values greater than or equal to 0.6, and fix the lower bounds on any such Booleans to 1 using the* **`thisOplModel.hireWorker[w].LB = 1`** *statement.*
4. *Undo the relaxation to the model using the* **`thisOplModel.unconvertAllIntVars()`** *command. Solve the model using the* **`cplex.solve()`** *command.*
5. *Use the previous solution as an advanced start to the original MIP. To do this, reset all lower bounds to 0 and solve the MIP.*
6. *Check your script with the solution in the* **`<trainingDir>\OPL63.labs\Staffing\Scripted_models\IP_Relaxation\solution`** *directory, and make any necessary changes.*
7. *Run your model and check the* ***Scripting log*** *output tab to follow the sequence of steps.*

IBM® ILOG® Script is especially useful for writing heuristics such as the one your just implemented. Without script, you would've had to write separate models, solve each of the models individually and fix variable values manually in between solves. IBM ILOG Script automates all these steps in the **`main`** *script block, without changing anything in the original OPL model definition.*

Model access script

Objective

- *Practice using flow control and model access to solve a variation of the original model.*

Actions

- *Problem description*
- *Steps to complete the model access script*

References

IloOplModel
script syntax

Exercise folders

`<trainingDir>\OPL63.labs\Staffing\Scripted_models\Model_Access\work`

You can find useful examples related to model access and flow control in the example directory installed with the product, **`<OPLhome>\examples\opl`***, where* **`<OPLhome>`** *is the top level directory where OPL is installed. Specifically,* **`mulprod`** *and* **`cutstock`** *projects, in the* **`main`** *run configurations, contain some examples for changing decision variables and accessing model information using IBM® ILOG® Script.*

Problem description

1. *The foreman hopes to hire fewer people by getting one of the workers to work a few extra hours.*
2. *Determine which person should have their availability raised to reduce the number of workers by one.*
3. *Determine the minimum raise in availability required to reduce the minimum number of workers by one.*
4. *Use a script to solve this problem and do not edit the original model definition.*

Steps to complete the model access script

1. *Import the project* **`Staffing3Work`** *in the work folder. Leave the* ***Copy projects into workspace*** *box unchecked. Study the part of the script that has been completed and note that a name,* **`hiring`***, has been assigned to* **`thisOplModel`***.*
2. *Under the comment that reads "***`// define script variables worker, slackVal and lowest`***", define the following script variables:* **`worker`** *to denote the worker with least slack in their availability,* **`slackVal`** *to denote the slack value of each* **`ctAvailability`** *constraint, and* **`lowest`** *to denote the lowest slack value among all* **`slackVal`** *with initial value of* **`Infinity`***.*
3. *Fill in the* **`for (var w in hiring.namesOfWorkers)`** *loop: To determine which worker should have their availability raised, iterate through the list of workers,* **`namesOfWorkers`***, and determine which worker's availability constraint,* **`ctAvailability`***, has the least slack. With each iteration, first find the value of the slack,* **`slackVal`***, using the expression* **`thisOplModel.ctAvailability[w].slack`***. Next test whether* **`slackVal`** *is less than the lowest slack found thus far,* **`lowest`***. If it is, assign* **`slackVal`** *to* **`lowest`***. Break out of the loop if a* **`slackVal`** *equal to zero is reached.*
4. *Define and assign a value to a script variable representing the* **`skillGroup`** *corresponding to the worker with the lowest slack.*

   Use the **`workerSkillGroup[worker]`** *array.*

5. *Define and assign values to a script variable* **`availabilityIndex`** *representing the element in* **`workerSkillGroupPairs`** *corresponding to the* **`skillGroup`** *and the* **`worker`***.*

   Use the **`find`** *keyword.*

6. *Use a* **`while`** *loop to iteratively add 1 unit to the worker's* **`endingAvailability`** *and then solve the model. The* **`while`** *loop should stop as soon as the objective value is 1 less than the original objective. The bulk of this loop has been completed. You need to insert the termination criteria, a statement to increase the* **`endingAvailability`***,*

and a statement to set the worker's availability to the new **`endingAvailability`** *value.*

7. *Include an extra termination criteria for the* **`while`** *loop, namely that the* **`endingAvailability`** *should not be more then 20 units greater than the* **`initialAvailability`***, to cover the possibility that increasing this particular person's availability will not be of any help. Add an error message to be written to the Scripting log in case this condition is reached.*
8. *Check you script against the solution provided in the* **`<trainingDir>\OPL63.labs\Staffing\Scripted_models\Model_Access\solution`** *directory. Make any necessary changes.*
9. *Run your model and check the output in the* ***Scripting log*** *output tab.*

Column generation with IBM ILOG Script

Learning objective
Learn how IBM® ILOG® Script can be used to implement column generation.

Key terms
- decomposition
- column generation

What is column generation?

Column generation is a decomposition technique often used to solve complex optimization models. This technique is especially useful when a problem has many variables (columns in the CPLEX® matrix), but relatively few constraints.

The basic idea is to decompose the original problem into a master problem and a subproblem. The subproblem uses a subset of the constraints to generate a set of variables (i.e. columns) to be used by the master problem. The master problem is then defined using this subset of variables, thus resulting in a much smaller model than the original.

The sub- and master problems can be solved in sequence to generate one solution, or iteratively to try and find an improved solution with each iteration.

In the example that follows, you'll see how IBM ILOG Script can be used to implement column generation for a configuration problem. An interesting feature of this implementation is that it uses both CP Optimizer and CPLEX.

Instructor note
If you are not familiar with the Vellino problem, please refer to **Language > Language User's Manual > The application areas > Applications of constraint programming > The vellino example (column generation)** in the OPL documentation to help you explain the example to the students.

Practice

Example: Column generation using both CPLEX and CP Optimizer

This configuration problem involves placing objects of different materials (glass, plastic, steel, wood, and copper) into bins of various types (red, blue, green), subject to capacity (each bin type has a maximum) and compatibility constraints. All objects must be placed into a bin and the total number of bins must be minimized.

Import the project from **`<OPLhome>\examples\opl\vellino`** and study this with your instructor.

The original model is decomposed into two models, namely an MP model (the master problem) and a CP satisfiability model (the sub problem).

The CP satisfiability model is used to generate all possible bin configurations, as most of the compatibility constraints are logical constraints for which the CP Optimizer engine offers good support.

These are then passed to the MP model, which is used to select the best combination of configurations.

The MP problem is a linear set covering problem where each decision variable corresponds to a possible configuration.

The **`cutstock`** example you looked at earlier uses column generation with an iterative approach.

Summary

Review

Among the benefits of using IBM ILOG Script in your models are:

- Possibility to repeat execution of multiple instances of models within a single script session
- Rich communication between script and models
- Access to model state information
- Rich data output features including database access

In this lesson, you learned how to implement flow control in an IBM ILOG Script **`main`** block using the extension classes for OPL. Specifically, you learned how to:

- Access and modify model and data elements
- Modify the CPLEX matrix incrementally
- Use looping controls
- Use a warm start
- Perform integer relaxation
- Execute postprocessing from a **`main`** block
- Debug a **`main`** block

You applied some of these techniques in the Staffing lab, and saw some flow control examples, including a column generation implementation.

Additional resources

Examples

- **`<OPLhome>\examples\opl\cutstock`**
- **`<OPLhome>\examples\opl\mulprod`**
- **`<OPLhome>\examples\opl\vellino`**

Tutorials

- **Language > Language User's Manual > IBM ILOG Script for OPL > Tutorial: Flow control and multiple searches**
- **Language > Language User's Manual > IBM ILOG Script for OPL > Tutorial: Flow control and column generation**
- **Language > Language User's Manual > IBM ILOG Script for OPL > Tutorial: Changing default behaviors in flow control**

Lesson 16: Integrating OPL Models with Applications

Instructor note
This lesson should last about 2 hours.

Once a model has been developed, debugged and fine-tuned, you will often want this model to interact with other, existing tools for analysis and control of your organization. To do this, IBM® provides a set of programming interfaces in OPL.

This lesson introduces the IBM ILOG OPL Interfaces, which give you full access to the CPLEX® and CP Optimizer APIs from external applications.

At the end of this lesson, you will:

- Know when to use the OPL interfaces to integrate an OPL model into an application
- Have an overview of the interfaces and how they work
- Know how to program a basic integration module for a simple OPL model

The process of OPL model integration

Learning objective
Understand how OPL communicates with the outside world, and when it is best to use the OPL programming interfaces.

Key terms
- development mode
- OPL application programming interfaces (APIs)

In order to integrate your model into an external application or suite of applications, the model needs to communicate with the outside world.

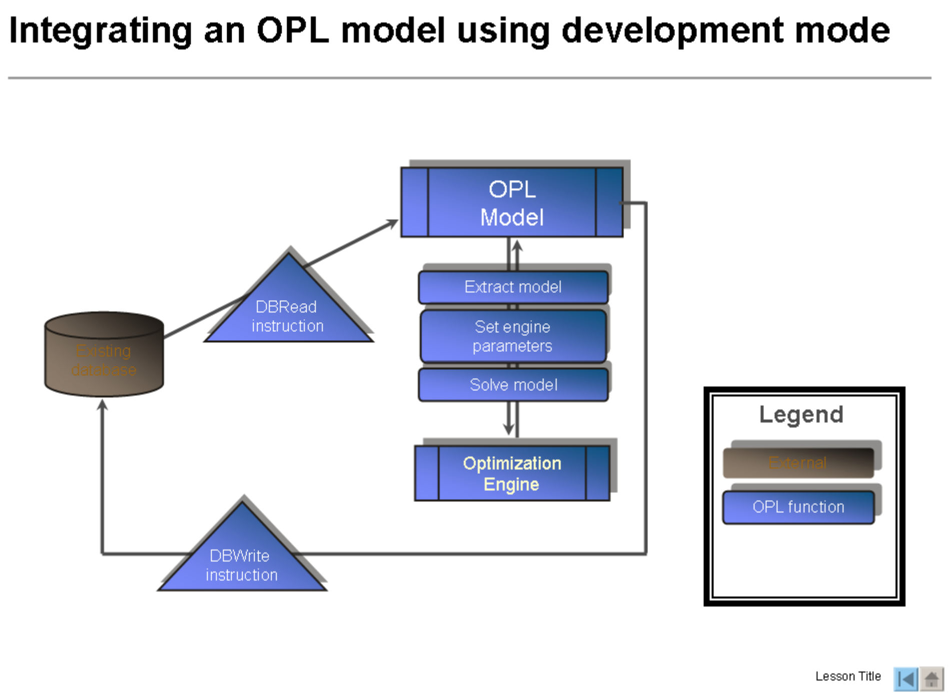
OPL communicates with the outside world in three ways:

- OPL Studio's development mode – i.e. via commands that read and write data in a spreadsheet or database
- OPL application programming interfaces (APIs)
- Via an ODM application.

The focus of this lesson is the use of the OPL programming interfaces. The other methods are presented in separate lessons.

Using OPL Studio development mode

The diagram below shows how OPL interacts with a database in development mode. You are limited to reading, writing and updating the database. Spreadsheets work similarly, except you can only read and write.



Details and syntax for this process have already been covered in an earlier lesson.

OPL communicates directly with the optimization engine to set engine parameters, extract and solve the model.

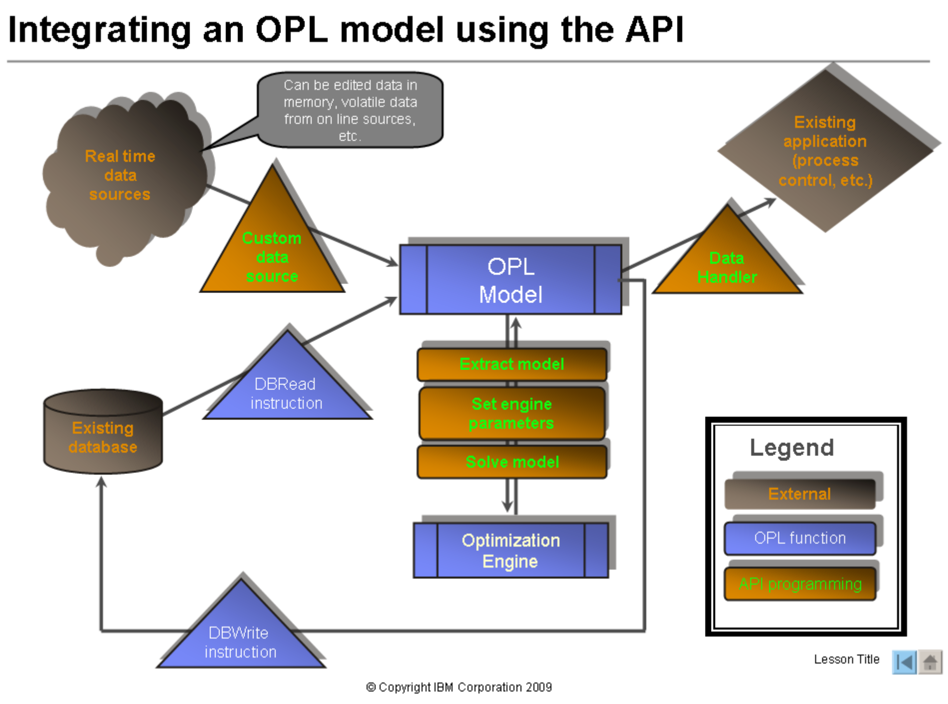
Using the OPL API

The diagram below shows a typical scenario for OPL interactions with a database and with other external applications via the API. In this scenario, the external program is responsible for managing data imports and exports to external programs other than the database.

It may also, optionally, handle:

- Model extraction
- Setting engine parameters
- Running the solve

The advantage of doing these things with the external program is that the OPL project itself contains only the modeling information. The model is instantiated at runtime, and the engine control processes are handled by the integration program.



When to use the API

Some examples of when you opt for the OPL interfaces include:

- You need to communicate with a different type of application than a spreadsheet or database
- You have an existing application with its own interface that everyone knows, and you want to integrate the model into that interface
- You need to work with volatile data, coming from real-time sources such as the internet, or calculated by another application and held in program memory

Using the interfaces, you can write modules that interact with the OPL engines in these programming languages:

- C++
- Java

- .Net languages such as:
    - VB.NET
    - C#
    - etc.

These interfaces allow you to:

- Solve a model many times with different parameters
- Solve several OPL models
- Modify a model.

The IBM ILOG OPL Interfaces

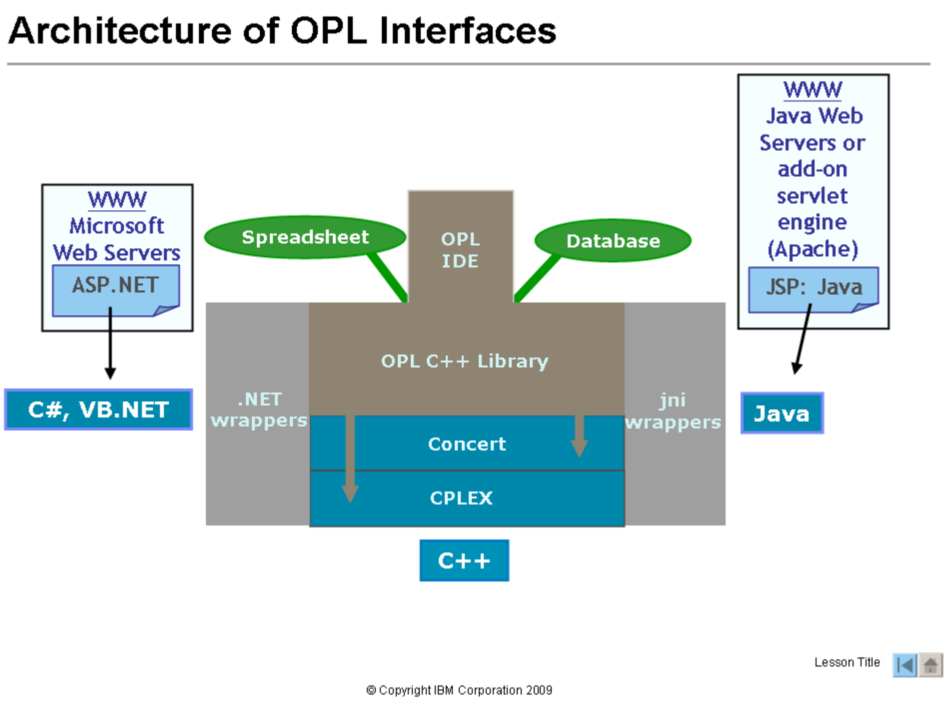
Learning objective
Gain familiarity with the 3 programming APIs available to work with OPL

Key term
IBM ILOG OPL Interfaces

The IBM® ILOG® OPL Interfaces provide 3 different APIs:

- The .NET API:
  - Provides interaction with Visual Basic.NET, C#
  - Refers to the **`oplall.dll`** and **`opl<versionNumber>_dotnet.dll`** dynamic library
- The generic C++ API links with many libraries placed in **`<OPLhome>\lib\<port_name>\format`**
- The Java Native Interface (JNI) API refers to the **`OPL61.dll`** dynamic library

The following diagram shows the architecture of the IBM ILOG OPL Interfaces:



Importing the libraries

Each API needs to import a set of libraries to use the OPL Interfaces:

- For Microsoft® .NET:
  - **`ILOG.Concert`**
  - **`ILOG.CPLEX`**
  - **`ILOG.CP`**
  - **`ILOG.OPL`**
- For Java™:
  - **`ilog.concert.*`**
  - **`ilog.cplex.*`**
  - **`ilog.cp.*`**
  - **`ilog.opl.*`**

- For C++:
  - **`<ilopl/iloopl.h>`**
  - **`<ilcplex/ilocplex.h>`**
  - **`<ilcp/cp.h>`**

Practice

Demonstration: A simple example
You are now going to take a look at a working example of some of the different ways the APIs can be used. Your instructor will guide you through the steps, which are also described here in the workbook.

The files for demo can be found with the labs for this training:

`<training_dir>\OPL 63.labs\APIModelIntegration`

These examples are programmed using Java^{TM}. Similar results can be obtained with the other interfaces.

The problem is a simple staffing allocation problem that includes:

- Employees
- Daily cost per employee
- Slots to which employees can be assigned
- Couplings of slots and employees which are not allowed

The objective is to minimize costs subject to meeting the demand.

Instructor note
This is intended as an instructor-led demo, not a hands-on lab. There are no slides to accompany this demo, as it is presumed you will put your desktop on the screen to show how to do it.

1. Show the students the OPL project (found in **`APIModelIntegration\oplmodel`** with its two run configurations. There should be nothing in this model that is not already familiar to them.
2. The next step is to show how to write code to call this model from Java
3. In the last step, you also demonstrate how to read the data from a custom database.

Inside the **`APIModelIntegration`** directory, the **`data`** directory contains excel spreadsheets used for reading input data and writing results. There are also three **`.csv`** files that are the simulated data sources for the custom data source class.

The **`java`** directory contains a **`src`** subdirectory where you can find the source code for the Java classes used in this demo, and a **`classes`** subdirectory where you will find built classes that can be used to run the application. You can use an **`ant`** script to launch the application:

- Run **`ant oplmodeljavamainrunxlsdata`** from the **`java`** directory to run the basic OPL model without **`main`** block, that reads data from an Excel spreadsheet.
- Run **`ant oplmodelscriptmainrunxlsdata`** from the **`java`** directory to run the OPL model controlled by an OPL Script **`main`** block, that reads data from an Excel spreadsheet.
- Run **`ant oplmodeljavamainrunjavadata`** from the **`java`** directory to read the custom data source and run the OPL model

You can also run these scripts from Eclipse, or another Java IDE, by loading the **`build.xml`** file.

In Eclipse, the procedure is:

1. Load the **`build.xml`** file into the IDE.
2. In the **Outline**, right click the class you want to execute.
3. Select **Run As > Ant Build** from the context menu.
4. The output is displayed in the **Console** output tab.

Comparison with an all-OPL solution

In this example, the **`oplmodel`** project contains two versions of this problem that use only the OPL IDE to solve the model. Use these for comparison with the java-based solutions:

- The run configuration **OPLIDEMainXLSDataFileConfig** uses data read from a spreadsheet, with IBM® ILOG® Script used for preprocessing and postprocessing.
- The run configuration **ILOGScriptMainXLSDataFileConfig** is identical with the addition of the model file, **`opldmainmodel.mod`** containing an IBM ILOG Script **`main`** block, used to control each step of the problem's execution.

It is important to note the use of the OPL keyword **`include`** in this model, which permits modular reuse of different parts of the model for different run configurations and for the java-based examples, as well.

Invoking an OPL .mod file from Java

Now suppose that this OPL model needs to be included in a suite of tools used by an organization to plan its activities. You can see how to do this in Java by looking at the source files found in the **`APIModelIntegration\java\src\javaoplapi`** directory.

The **`Java OplModelRun.java`** source code creates a class that simply calls the OPL model and executes it when necessary. This code looks for an IBM ILOG Script **`main`** block, and if it is present, lets the script control the problem's processing and solving steps. Otherwise, the java class controls them. The code also includes error handlers.

It can also be the case that your input data is already memory resident in a data and/or business object model, or that it needs to be read from specific data sources that are not Data Bases, Excel spreadsheets or OPL **`.dat`** files (for example, XML, in-house applications, business process managements systems, etc.). In that case, you need to create a custom data source in Java:

- The **`CsvDataSource.java`** source code file creates a custom data source class that simulates (in this case) reading the data from such a source. In this simulation, it reads the data from a **`.csv`** file provided in the **`APIModelIntegration\data`** directory.
- The **`JavaApiOplModelRun.java`** source code file creates a class that reads the data using the **`CsvDataSource.java`** data source and instantiates the same OPL model with the data it reads.

Examples using the other interfaces

You can see examples of the same problem modeled in each of the APIs. The statements for importing the libraries are found at the top of each source file. This is the stock cutting problem you looked at in the previous lesson.

Open the following files in the OPL IDE editor or other editor:

- For Java™: **`<OPLhome>\examples\java\cutstock\src\cutstock\Cutstock.java`**
- For VB.NET: **`<OPLhome>\examples\dotnet\Cutstock\Cutstock.vb`**
- For C++: **`<OPLhome>\examples\cpp\src\cutstock.cpp`**

Examine how the same problem is modeled in each API.

OPL extension classes

Learning objective
Gain practice via a hands-on tutorial with some of the OPL extension classes

Key term
OPL extension classes

The examples you have been looking at all use the **OPL extension classes**.

IBM provides major extension classes for use with Java^{TM}, .NET, and C++ to access, modify, and run OPL models. The extension classes vary in their naming conventions, but share the same methods and attributes.

In general, The C++ and Java class names start with **`IloOpl`**, while the .NET class names start with **`Opl.`**

Finding the class reference documentation

The documentation of all the classes of the OPL APIs can be found in the documentation, in the *Interfaces* manual:

- *Interfaces User's Manual*
- *C++ Interface Reference Manual*
- *Java Interface Reference Manual*
- *.NET Interface Reference Manual*

You will find complete, up-to-date documentation of all the classes and their methods in these manuals.

Practice

Extension classes tutorial

Go to the **OPL Interfaces Tutorial** in the workshop. Go through all of it with your instructor. It will present 8 of the 10 extension classes and their methods.

The oplrun command

Learning objective
Learn how oplrun can be used to run models and projects from the command line, and how compiled model files can be used to connect with the programming APIs

Key terms
- oplrun command
- .opl file

OPL allows you to execute a model or project from the command line (Windows or UNIX environments) using the **`oplrun`** command.

Executing a model from the command line

You can execute a **`.mod`** file as a standalone model using the following syntax:

```
oplrun [<options>] <model-file> [<data-file> ...]
```

The **`<options>`** and **`<data-file>`** arguments are optional. However, if the model requires external data, the **`<data-file>`** argument is mandatory. For a list of options, refer to **oplrun Reference> Running OPL from the Command Line** in the **Other Reference** documentation.

Standalone models are no longer supported in the IDE from OPL 5.0 on. In general, it is preferable to run a project, however, running standalone models presents no problems when using **`oplrun`**.

Executing a project from the command line

You can run a project from the command line, and can specify a run configuration to use during execution. Use the following syntax:

```
oplrun [<options>] -p <project-file> [<run-configuration>]
```

The **`<options>`** and **`<run-configuration>`** arguments are optional. If no run configuration name is specified, the default configuration is executed.

Compiled models

OPL provides the possibility to generate compiled model files from within the IDE. A compiled model is a binary version of a **`.mod`** file.

You can pass a compiled model file name as an argument to a method of the OPL Interface libraries. An additional benefit of keeping your models as compiled files is that these binary files are not human-readable, and this may help you protect your intellectual property.

A compiled model can also be run from the command line, without starting the OPL IDE, using the **`oplrun`** command with the **`-c`** option. Refer to **oplrun Reference> Running OPL from the Command Line** in the **Other Reference** documentation.

A compiled model (**`.opl`**) cannot be loaded directly into the IDE

Practice

Create a compiled model from a project

Perform the following steps:

1. Open a validated project in the IDE in the usual way.
2. Rick click on a model file and select **Compile Model** in the contextual menu The corresponding compiled model is generated. The extension of the generated file is **`.opl.`**

   For example, if you generate a compiled model file for **`gas2.mod`**, the resulting file will be **`gas2.opl.`** By default, the file is saved in the same directory as the model file.
3. Click Save and check for the compiled model file (for example, in the Windows Explorer).

Practice

You are now going to perform 2 workshop steps. They are similar to the IBM ILOG Script steps you've already done, except they're done in OPL Interfaces. The basic concepts behind the workshops do not change, but the language required to access the OPL data structures does.

Each workshop step is available for 2 interfaces (Java and .NET): just select your preferred API. Each workshop step uses one or more examples from the examples folder. You can copy from these examples to create the solution.

The first workshop step is just to get comfortable with the Interfaces and model building. The second workshop step goes deeper into the OPL data structures.

1. Perform the **Staffing problem > LP relaxation API** workshop step.
2. Perform the **Staffing problem > Model access API** workshop step.

You can perform this lab using the HTML workshop, or by following the instructions in the workbook. The HTML workshop will give you direct access to OPL documentation pages that can help you with the lab.

LP relaxation API

Objective

- *Get familiar with using the OPL Interfaces.*

Action

- *LP relaxation steps*

Reference

IloOplModel in Java

Exercise folders

`OPL63.labs\Staffing\API\LP_Relaxation\Java\work`

`OPL63.labs\Staffing\API\LP_Relaxation\dotnet\work`

When working in Visual Studio, add references to the appropriate .dll as follows:

1. *Open the **`.sln`** file*
2. *Go to the **Solution Explorer***
3. *right click the project or the **References** folder*
4. *Check that the OPL dll (**`oplall.dll`**) is in the references, and if not add it by selecting **Add References***

*You can view the **`mulprod`** example in the **`<OPLhome>\examples\...`** folder to see similar code as some of the code used for this lab.*

LP relaxation steps

In this lab, you will duplicate the LP relaxation script step using the API language code. In this version, you will first solve the original model, then the relaxed model, and then revert back to the original model.

1. *Open the appropriate file in the work folder, depending on the API you choose. Study the code that has been partially completed.*
2. *Use the **`convertAllIntVars()`** method to convert the model into the relaxed model.*

3. *Use the* ***`unconvertAllIntVars()`*** *method to convert the model back to a MIP.*
4. *Call* ***`postProcess()`*** *where indicated and note the effect thereof when solving the model. This demonstrates how tasks that are more easily programmed in the .mod file can be written in the post processing script and called in the API code.*
5. *Cross-check your code with the solution in the* ***`solutions`*** *folder, and make any necessary changes.*
6. *Run your model.*

Model access API

Objective

- *Use flow control and model access to increase worker availability and reduce the number of workers required*

Action

- *Steps to completing the lab*

References

IloOplModel in Java
IloForAllRange
IloCplex
IloTupleSet
IloOplDataElements
IloSymbolSet

Exercise folders

`OPL63.labs\Staffing\API\Model_Access\Java\work`

`OPL63.labs\Staffing\API\Model_Access\dotnet\work`

Examples that use similar code as used for this lab, are ***`cutstock\cutstock_change.mod`*** *and* ***`mulprod\mulprod_main.mod`*** *in the* ***`<OPLhome>\examples\opl`*** *folder.*

Steps to completing the lab

1. *Open the appropriate file in the work folder, depending on the API you choose. Study the code that has been partially completed.*
2. *Fill in code to link the program to* ***`staffing3.mod`*** *and* ***`staffing3.dat`*** *in the work folder for your chosen API.*
3. *Add code to solve the model and store the objective value.*
4. *Open* ***`staffing3.mod`*** *file for editing and scroll down to the post processing script. Note that the post processing is now used to both print the solution, and also to write the slack values for the* ***`ctAvailability`*** *constraints to an array. You will access the slacks in the API using this array. In your chosen API program, call* ***`postProcess`*** *after solving the model.*
5. *Write the* ***`slacks`*** *from the array in the OPL model to an* ***`IloNumMap`*** *called* ***`workerSlackMap`*** *by using the* ***`getElement`*** *method. Note the various class names used to access OPL data types throughout the code.*
6. *In order to find the tuple corresponding to the skill group and worker with the least slack, get the contents of the* ***`skillGroupNames`*** *tuple set using the* ***`getElement`*** *method and iterate through the set until you've found the* ***`index`*** *that corresponds to the relevant skill group and worker.*

7. *Create a tuple from this **`index`** by using the **`makeTuple`** method, and find the **`initialAvailability`** associated with that tuple from the **`workerAvailabilityMap`**.*
8. *In the **`while`** loop, create new **`IloOplModel`** and **`IloCplex`** instances (see the top of the code for examples).*
9. *After studying the remaining code, compare your code with that found in the solutions directory and run the model.*

Summary

Review

In this lesson, you learned about IBM ILOG OPL Interfaces:

- .NET, Java and C++ libraries allow the integration of OPL models within applications.
- You can generate compiled model files, and use these models as arguments for applications via the interfaces.

Lesson 17: Optimizing Engines and Algorithms

When you have a big problem to solve, involving large numbers of decision variables and constraints, the calculation process can be long. Naturally, you want to be sure you are using the most efficient means to arrive at a quick, reliable solution. OPL is built on IBM®'s proven technology. The CPLEX® and CP Optimizer engines provide different strategies for Linear Programming (LP), Mixed Integer Programming (MIP), Quadratic Programming (QP) and Constraint Programming (CP) to assure the fastest calculation time possible for your particular problem.

Instructor note
This lesson should last about 1 hour 15 minutes.

You must first choose whether you want to use CPLEX or CP Optimizer to solve the problem.

Then, depending on this choice, you could optionally make some additional choices related to the solve process, such as specifying the algorithm used by CPLEX, setting some solver parameter settings, or specifying a search phase for CP.

This lesson provides:

- Guidelines for choosing which engine to use
- An overview of the different MP optimization algorithms available in IBM ILOG OPL
- Guidelines for deciding which MP algorithm to choose
- Guidelines for fine tuning the algorithms in the settings file

OPL provides two powerful engines: CPLEX and CP Optimizer.

CP Optimizer uses Constraint Programming techniques, featuring built in search. The built in search can also be customized if necessary.

CPLEX provides access to seven different MP algorithms:

- Simplex optimizers can solve linear and quadratic problems with millions of constraints and continuous decision variables:
  - Primal
  - Dual
  - Network (a specialization of the primal and dual simplex optimizer for network problems)
- Barrier optimizer, an alternative to the simplex method for solving linear and quadratic problems, and an approach for solving quadratically constrained problems (QCP)
- Mixed Integer Programming (MIP) optimizer for problems with mixed-integer decision variables (continuous or binary) using linear or quadratic objective functions
- Sifting optimizer
- Concurrent optimizer

These optimizers are described briefly in this lesson, followed by information on configuring them and guidelines for selecting them.

Choosing your optimization engine

Learning objective
Learn the basic guidelines for choosing the correct engine.

Key terms
- CPLEX
- CP Optimizer

The first decision you need to make, is whether the model you are about to write is a candidate for MP or for CP. You need to make this decision at the start, as the design of the model is intrinsically different, most of the time.

CPLEX® is used as the default optimization engine if you do not specify one. Select CP Optimizer by typing the command **`using CP`** as the first line in your model (**`.mod`**) file.

Guidelines for choosing your engine

You need to make your choice of engine using your knowledge of the problem you are going to model. Here are a few guidelines that can help:

Use CPLEX when:
- The problem can be expressed naturally using MP
- You need proven optimality, or a robust measure of the percentage of optimality of the best solution found

Use CP Optimizer when:
- You need to do detailed scheduling
- The problem contains numerous or complex logical constraints
- The problem contains constraints that are hard to express using MP (e.g. quadratic expressions, generalized elements, standard deviation, lexicographic ordering, allowed/forbidden assignment combinations, etc.)

Very large problems, which may require different engines to solve different aspects, should be decomposed into multiple models, and each engine used for the appropriate part of the problem. For example, planning using CPLEX, and detailed scheduling of the resulting plan using CP Optimizer.

It is important to remember that these are general guidelines, and your knowledge of the problem, together with accumulated experience, will provide the best guidance. If you have a problem that does not use expressions that only one of the engines can extract, it is worthwhile to try both engines to see which is more efficient in solving your problem.

CPLEX optimization algorithms

Learning objective
Learn the different MP optimizing algorithms available in the CPLEX® engine, and how each works

Key terms
- simplex
- barrier
- MIP
- sifting
- concurrent
- CP

The Primal Simplex optimizer

The Primal Simplex optimizer has two phases:

- A first phase finds a feasible starting point.
    - The starting point is an extreme point belonging to the edge of the feasible region.
    - If you do not provide an objective function in your model, OPL gives the starting point as the solution.
- A second phase iteratively improves the optimality.

The Primal (and Primal Network) Simplex algorithm reaches optimality by following the edge of the feasible region.

The Dual Simplex optimizer

The Dual Simplex optimizer has two phases:

- A first phase finds an optimal but not feasible starting point.
- A second phase iteratively improves the objective toward optimality.

Dual Simplex is generally faster than Primal Simplex when the number of constraints exceeds the number of decision variables but this is very sensitive to problem structures, particular cases, etc.

The Network Simplex optimizer

The Network Simplex optimizer includes adaptations of the Primal and Dual Simplex algorithms to problems having a network substructure. It functions similarly to Primal and Dual, with the addition of these adaptations.

The Barrier optimizer

The Barrier optimizer is a one-phase interior point method reaching optimality and feasibility simultaneously. It can generate solutions inside (suboptimal), outside (infeasible) or on the edge of the feasible region, using a primal-dual, predictor-corrector method to converge toward optimality. The Barrier optimizer is generally more efficient than the Primal Simplex for large size problems (thousands of decision variables and constraints). It also offers a fast, robust method for solving quadratically constrained programs.

The MIP optimizer

The MIP optimizer employs a branch-and-bound technique that can be used to solve mixed-integer linear programs (MILP); mixed-integer quadratic programs (MIQP); and mixed-integer quadratically constrained programs (MIQCP). This optimizer includes cutting-plane strategies such as Gomory, clique and cover, flow cover, GUB cover and implied bound. Cutting plane strategies are beyond the scope of this training, but more information can be found in the CPLEX user documentation.

The MIP optimizer is automatically invoked in problems using CPLEX when there are integer decision variables or data in the model.

The Sifting optimizer

The sifting optimizer performs best when the problem has a large aspect ratio. That is when there are many more decision variables than constraints. It works well in problems where an optimal solution can be expected to place most decision variables at their lower bounds.

Sifting optimizer process:

1. It first solves the working problem, i.e. a subproblem:
    - All rows are included.
    - A small subset of the columns is included.
    - The remaining columns are assumed to have an arbitrary solution, such as decision variables fixed at their lower bounds.
2. It uses the solution of the working problem to re-evaluate the reduced costs of the remaining columns.
3. Any such columns whose reduced costs can improve the objective of the working problem become candidates to be added to the working problem for the next major sifting iteration.
4. When no candidates are present, the solution of the working problem is optimal for the full problem. Sifting terminates.

The Concurrent optimizer

The concurrent optimizer can be used with a parallel processing server and a parallel (multi-thread) CPLEX license. It solves the problem by simultaneously using Primal Simplex, Dual Simplex, and Barrier algorithms and returns a solution when any of the algorithms finds a solution.

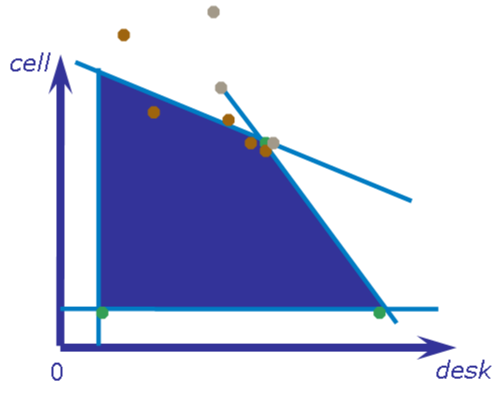
The concurrent optimizer launches distinct optimizers on multiple threads. When the concurrent optimizer is launched on a single-threaded platform, it calls the Dual Simplex algorithm.

In other words, choosing the concurrent optimizer makes sense only on a multiprocessor computer where threads are enabled.

MP Optimization strategies compared

The following 2 diagrams illustrate how the most commonly used MP optimizers function. The first diagram shows where each optimizer looks for an initial solution or starting point in its first phase or cycle.

Optimizer starting points

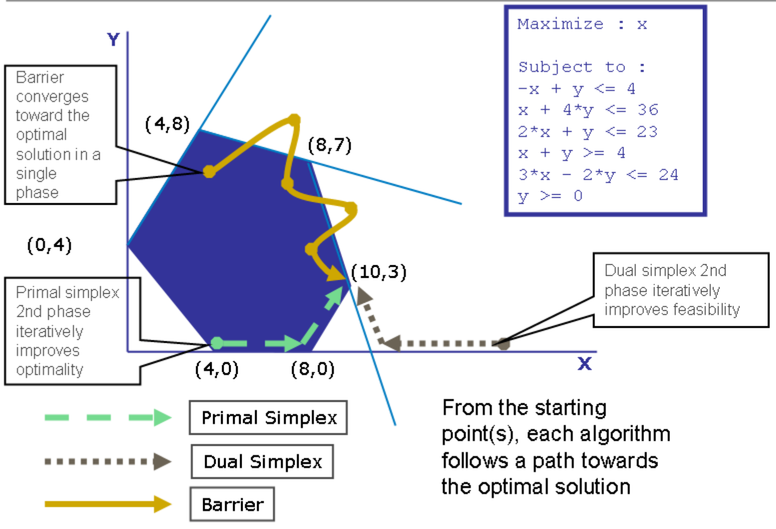


Algorithm	1st phase	2nd phase
● Primal Simplex	selects a starting point that is on the edge of the feasible region	iteratively improves optimality
● Dual Simplex	selects a starting point that is optimal but not feasible	iteratively improves feasibility
● Barrier	selects multiple starting points, both inside and outside the feasible region and converges toward the optimal solution in a single phase	

Lesson Title

The next diagram shows how the optimizer goes about moving toward its optimal solution.

Some LP Optimizer strategies


Y
X
(4,8)
(8,7)
(0,4)
(10,3)
(4,0)
(8,0)
Barrier converges toward the optimal solution in a single phase
Primal simplex 2nd phase iteratively improves optimality
Dual simplex 2nd phase iteratively improves feasibility
Primal Simplex
Dual Simplex
Barrier

```
Maximize : x

Subject to :
-x + y <= 4
x + 4*y <= 36
2*x + y <= 23
x + y >= 4
3*x - 2*y <= 24
y >= 0
```

From the starting point(s), each algorithm follows a path towards the optimal solution

Controlling optimization

Learning objective
Learn how and when to configure optimizing algorithm options

Key terms
- settings file
- controlling optimization

OPL gives you a large measure of control over the selection of optimizers and the configuration of their parameters. You perform both of these operations in the settings (`.ops`) file.

This topic steers you to the configuration options in the settings file, and gives you some guidelines for choosing the best algorithm for a given problem. A project does not automatically have a settings file, but you can create one at any time.

You can also configure the optimizer parameters using IBM® ILOG® Script.

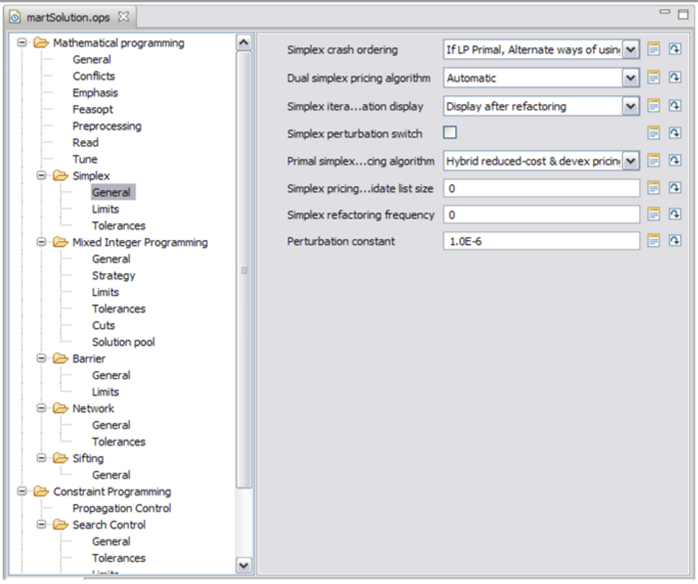
Changing default parameters

For most problems, the default parameters function best. There are occasions, however, where you want or need to change certain parameters to control the behavior of the optimizer. The theory and practice of fine tuning optimizer performance is an advanced topic and beyond the scope of this training. Here you will learn how parameters are modified, should you wish to do your own performance tuning.

To change parameters of an algorithm:

1. Create a new settings file by selecting, from the main menu, **File >New >Settings.** After choosing the folder and file name for saving, the settings file opens in the OPL IDE workspace.
2. Select a group of parameters where you want to make one or more modifications. For example, select the **General** leaf of the **Simplex** branch of the navigation tree. The window resembles the following illustration:

General Simplex parameters



Simplex crash ordering	If LP Primal, Alternate ways of usin
Dual simplex pricing algorithm	Automatic
Simplex itera...ation display	Display after refactoring
Simplex perturbation switch	☐
Primal simplex...cing algorithm	Hybrid reduced-cost & devex pricin
Simplex pricing...idate list size	0
Simplex refactoring frequency	0
Perturbation constant	1.0E-6

Lesson Title

© Copyright IBM Corporation 2009

In this window, you can select parameters you wish to modify. Some parameters are selected from a list, others can be entered directly from the keyboard. Each optimizer has its specific set of parameters. You can see a quick popup description of the parameter's function by hovering your mouse cursor over the “note” icon alongside the parameter.

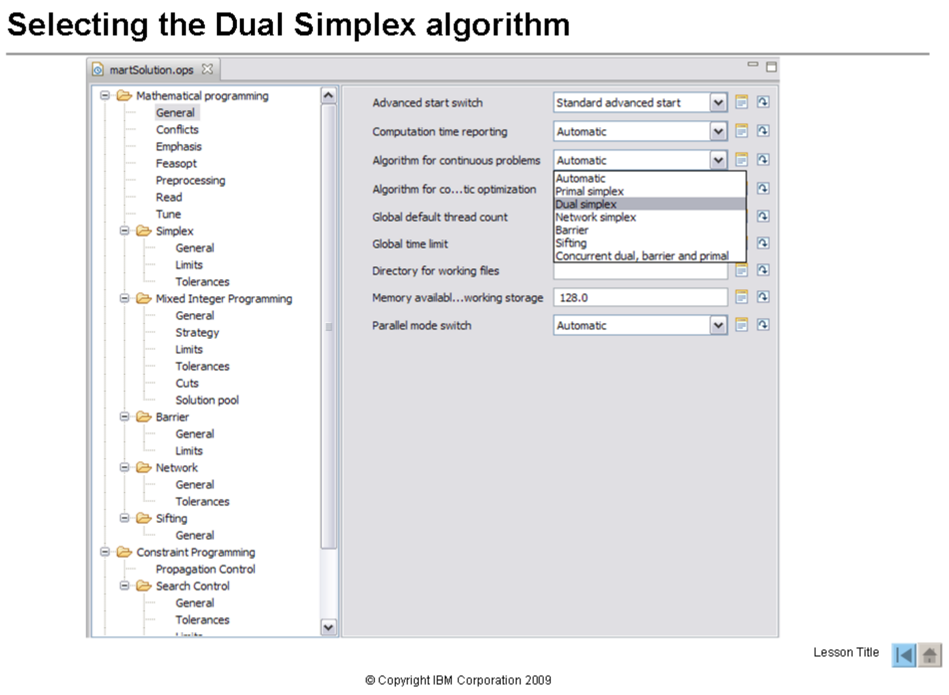
The curved arrow icon, if clicked, restores the default setting for the parameter.

Selecting your optimization algorithm

There are several places where you can decide either to force the use of a particular algorithm or let the engine decide automatically, for example:

- Select algorithms to use for continuous problems and for continuous quadratic problems in the **General** leaf of the **Mathematical Programming** branch in the navigation tree of the dialog box.
- Select algorithms used for different processes of the barrier optimizer in the **Barrier >General** leaf.
- Select the algorithm used to solve the sifting subproblem in the **Sifting >General** leaf.

The following screen shot shows how you select the dual simplex algorithm for linear optimization.



Selecting the Dual Simplex algorithm

martSolution.ops

- Mathematical programming
  - General
  - Conflicts
  - Emphasis
  - Feasopt
  - Preprocessing
  - Read
  - Tune
  - Simplex
    - General
    - Limits
    - Tolerances
  - Mixed Integer Programming
    - General
    - Strategy
    - Limits
    - Tolerances
    - Cuts
    - Solution pool
  - Barrier
    - General
    - Limits
  - Network
    - General
    - Tolerances
  - Sifting
    - General
- Constraint Programming
  - Propagation Control
  - Search Control
    - General
    - Tolerances

Advanced start switch	Standard advanced start
Computation time reporting	Automatic
Algorithm for continuous problems	Automatic
Algorithm for co…tic optimization	
Global default thread count	
Global time limit	
Directory for working files	
Memory availabl…working storage	128.0
Parallel mode switch	Automatic

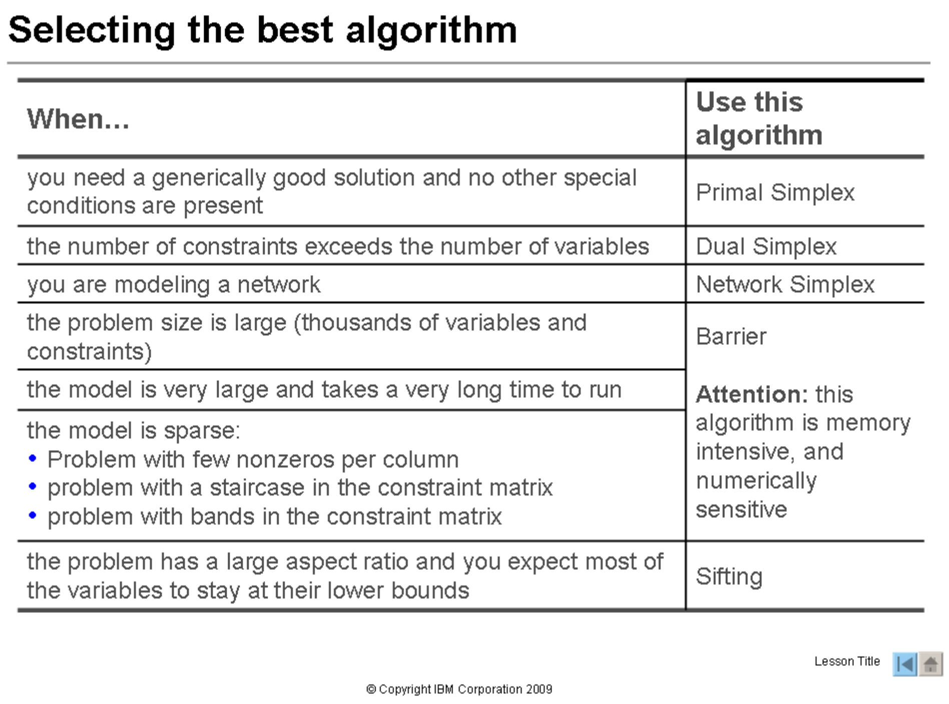
Automatic
Primal simplex
Dual simplex
Network simplex
Barrier
Sifting
Concurrent dual, barrier and primal

If you select **Automatic**, OPL will choose the algorithm for you. This is the default setting, but you can choose a specific algorithm from the list if you need to.

Your choice of algorithm and specific parameters apply only to the current active project and run configuration. Each run configuration can thus have its own customized settings, and optimizer behavior in different projects can be separately configured.

Selecting the best algorithm

The choice of algorithm is very sensitive to problem structure, particular cases and your specific data. Your knowledge of the problem and experience with OPL will always be the most important factors. The following table offers a few guidelines to help you make the choice.



Selecting the best algorithm

When…	Use this algorithm
you need a generically good solution and no other special conditions are present	Primal Simplex
the number of constraints exceeds the number of variables	Dual Simplex
you are modeling a network	Network Simplex
the problem size is large (thousands of variables and constraints)	Barrier **Attention:** this algorithm is memory intensive, and numerically sensitive
the model is very large and takes a very long time to run	
the model is sparse: • Problem with few nonzeros per column • problem with a staircase in the constraint matrix • problem with bands in the constraint matrix	
the problem has a large aspect ratio and you expect most of the variables to stay at their lower bounds	Sifting

Lesson Title

The concurrent algorithm can be useful if you have a multiprocessor machine and can work in multithreading mode.

Summary

Review

In this lesson, you learned about:

- Optimizing engines used by OPL:
    - CPLEX for Mathematical Programming
    - CP Optimizer for Constraint Programming

  You select your optimizing engine before writing the model.

- Optimization algorithms used by CPLEX:
    - Primal simplex
    - Dual simplex
    - Network simplex
    - Barrier
    - Sifting
    - MIP
    - Concurrent

  MP algorithms can be selected and configured in the settings file.

You have also learned some general rules of thumb for when to use which MP algorithm.

Lesson 18: Performance Tuning

Performance tuning focuses on two areas: speed and memory. OPL model resolution time can be greatly improved by paying attention to some syntax and modeling tips that deal with these issues.

Instructor note
This lesson should last about 30 minutes.

This lesson reviews some tips which, if followed, will bring significant results, due to:

- Better memory management
- Choice of syntax that works most efficiently with OPL's internal mechanisms
- Use of sparse model data, which reduces the number of elements that need to be calculated.

Many of these tips summarize material which you have already seen in the preceding lessons.

Use the **Profile** output tab in the OPL IDE to test different methodologies and fine tune your models.

Prefer declarative syntax

Learning objective
Improve the efficiency of OPL models

Key term
declarative syntax

In OPL, the rule of thumb is that declarative syntax is preferable to imperative syntax. For example,

```
int T[r in 0..10] = 2*r;
```

is better than

```
execute {
    for (r in 0..10)
  T[r] = 2*r;
 }
```

Use sparse arrays

Learning objective
Improve the efficiency of OPL models

Key term
sparse arrays

As in all programming languages, it is better to keep the data and model information as small as possible. In situations where there are fewer values than there are combinations for those values, it is better to use arrays of tuples. Thus,

```
tuple namesAndPlacesType {
  string name;
  string place;
 }
 {namesAndPlacesType} namesAndPlaces = ...;
 int availability[namesAndPlaces] = ...;
```

is better than

```
{string} names = ...;
{string} places = ...;
int availability[names][places] = ...;
```

If we used the two-dimensional array, we would have lots of zeros in the matrix – i.e. memory locations used by combinations which have to meaning for solving the problem. The array of tuples produces only those combinations that are non-zero and thus are of interest for solving the problem.

Think about data instantiation

Learning objective
Summarize hints related to data instantiation methods

Key term
instantiation

Throughout this training you have learned about different ways to instantiate data elements. Some of the advice may seem contradictory. In reality, the method you need to use is a function of your situation, the type of model you are constructing, and where bottlenecks may occur. Here you will find a number of tips assembled together, that present different considerations. It is impossible to follow all of them in one model. The objective is to select the method that is best for your situation.

Ranges and sets

Sets instantiated by ranges are represented explicitly (unlike ranges). As a consequence, a declaration of the form **`{int} s = asSet(1..100000);`** creates a set where all the values 1, 2, ..., 100000 are explicitly represented, while the range **`range s = 1..100000;`** represents only the bounds explicitly.

Pay attention to data initialization modes

In OPL, the initialization mode you choose affects memory allocation. In particular, external initialization from a **`.dat`** file, while enabling a more modular design, may have a significant impact on memory usage:

- Internally initialized data (directly from the model file) is initialized when first used. This is also called "lazy initialization". Unused internal data elements are not allocated any memory. In other words, internal data is "pulled" from OPL as needed.
- Externally initialized data (from a data file) is initialized while the **`.dat`** file is parsed and is allocated memory whether it is used by the model or not. In other words, external data is "pushed" to OPL.

As a general rule, you should maintain separation of model and data. Internal initialization should be considered only in cases of extreme memory use problems that cannot be overcome by other techniques.

Array instantiation - summary of tips

- Prefer generically instantiated indexed arrays rather than an **`execute INITIALIZE`** block – as is indicated earlier in this lesson, declarative syntax is more efficient, except for the most complex cases.
- Use sparse arrays – see above
- In pre- or postprocessing script statements, do not instantiate array elements to zero, OPL does that for you
- When you use multidimensional arrays, the order of the dimensions may be significant. For instance, in the following example:

```
/*..*/

range r1 = 1..n1;
range r2 = 1..n2;

dvar int+ x[r1][r2];

/*..*/

a1 == sum(i in r1, j in r2) x[i][j];
a2 == sum(j in r2, i in r1) x[i][j];
```

the calculation of **`a2`** is more efficient because the OPL internal caching mechanism recalculates **`x[i]`** only when **`i`** changes.

Scripting hints

Learning objective

Learn to use IBM® ILOG® Script efficiently

Key term

IBM ILOG Script

- Use the profiler to detect execute blocks that run for a long time during the preprocessing phase.
- If you observe that the execution of a model is slow because the main scripting block loads many engine instances or submodels, you can improve this by turning off the OPL Language option **Update charts and statistics in main** in the settings file.
- In pre- or postprocessing script statements, do not instantiate array elements to zero.
- Calculate iteration sets for conditional blocks.
- Declare local script variables using the keyword **`var`.**
- Use the methods **`end`** and **`endAll`** to free memory in IBM ILOG Script and interfaces.

Instructor note

The **`end()`** and **`endAll()`** methods are by default disabled in the IDE to prevent faulty memory management by the user. To enable them, we must set the **`mainEndEnabled`** setting on the model. See *OPL-2058* for more information.

Choose your MP optimizer

Learning objective
Improve the efficiency of OPL MP models

Key term
optimizer selection

Choose the MP optimizer based on the problem structure, and remember these basic guidelines:

- Use Primal Simplex for general problems.
- Use Network Simplex when the problem has a network substructure (each decision variable appears in two constraints).
- Use Dual Simplex when the problem has more constraints than decision variables.
- Use the Barrier Optimizer with large problems (thousands of decision variables and constraints).
- Use the Sifting Optimizer when the problem has more decision variables than constraints, and it is expected that most decision variables will remain at their lower bounds.
- Use the Concurrent Optimizer for very large or complex models when working in a multithreading environment

Additional performance tuning tips

Learning objective
Add to your bag of tricks for tuning models

Key term
performance tuning

- The general expression **`p in S`** where **`S`** is a set of tuples containing **`n`** fields, can be replaced by an indexed expression:**`<p1,...,pn> in S.`**

  While it speeds up processing, this process can be very memory consuming especially if used in labeled constraints! You need to evaluate which efficiency is most important for your model.

- Avoid dummy indexes for tuple components.
- In CP models with customized search strategies, consider the order of search phases.
- Change the way the engine solves models by changing engine parameters in the settings file.
- Constraint labels can result in large memory usage; when memory is a concern, constraint labels should be avoided.
- Often, memory usage and speed improvements are trade-offs, one against the other. Be sure you know which will give you the greatest improvement, and make your choices according to common sense.

Summary

Review

In this lesson, you learned some basic tips to improve the efficiency of your OPL models:

- Use declarative syntax
- Use sparse arrays
- Choose your optimizer based on the problem structure

Lesson 19: Appendix: The OPL IDE Graphical Interface

This optional lesson provides you with a quick overview of the OPL IDE graphical interface. It is not a complete tutorial. For that, please refer to the product documentation, where you will find tutorials and detailed explanations of the features and operation of the OPL IDE.

Instructor note
This lesson is optional. You can decide to present the user interface formally if you think the students will really need it (beyond what you show them during the regular lessons), or you can let them follow it on their own, as a self-guided tour through the interface. It can also serve as a reference guide during the training. Presented as a continuous lesson, it should take about an hour.

The OPL IDE main window

Learning objective
Learn the geography of the main OPL IDE window

Key terms
- OPL Projects Navigator
- Editing Area
- Output Area
- Problem browser
- Outline

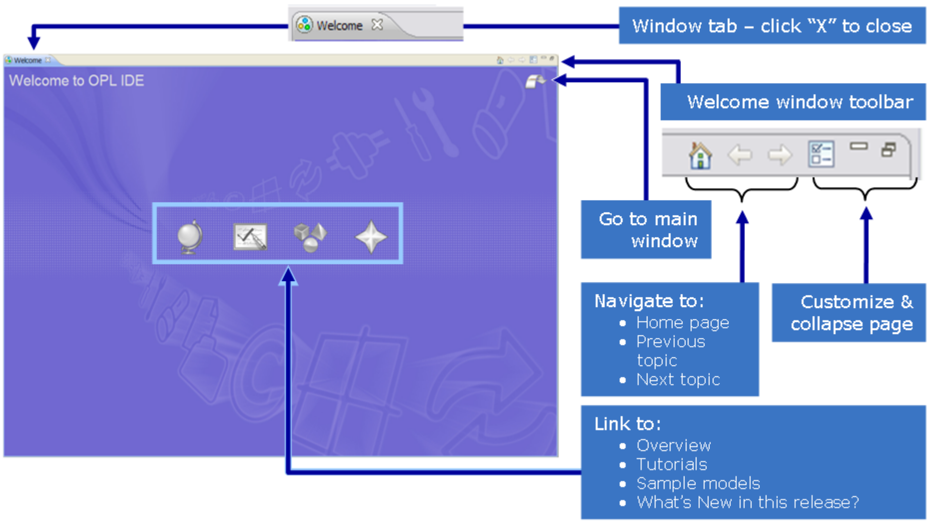
Starting the IDE

There are three ways to start the OPL IDE:

- From the Windows **Start** menu, click **Programs > IBM ILOG > IBM ILOG OPL > OPL IDE**
- In Windows Explorer, double-click the IDE executable **`oplide.exe`** in the **`<OPLhome>\oplide directory`**, where **`<OPLhome>`** is your installation directory
- From the command line, by typing **`oplide`**.

When the OPL IDE starts, by default you see the Welcome window:

The OPL Welcome Window



Lesson Title

© Copyright IBM Corporation 2009

The Welcome window provides a convenient means of navigating to different tools that help you learn more about the OPL IDE. These are identified by icons:

-

  Overview takes you to two parts of the Starting Kit documentation:
  - *From OR to OPL and ODM* provides an overview of IBM® ILOG® OPL and its main features.

    - *Migrating from previous versions of OPL* is useful for people who have used previous versions of OPL and need to migrate existing projects.
- * **Tutorials** provides a list of links to tutorials on various aspects of the OPL IDE
- **Samples** links you to sections of the documentation that explain the example files provided with the software
- **What's New** takes you to release notes that cover changes in the OPL IDE and in the underlying software, and also shows live links to the latest discussions in OPL and ODM user forums.
- The **Workbench** icon in the upper righthand corner of the Welcome window takes you to the OPL Main window.

The Welcome window has its own toolbar, used to navigate among the different elements available via this window, and to customize the way the page is displayed.

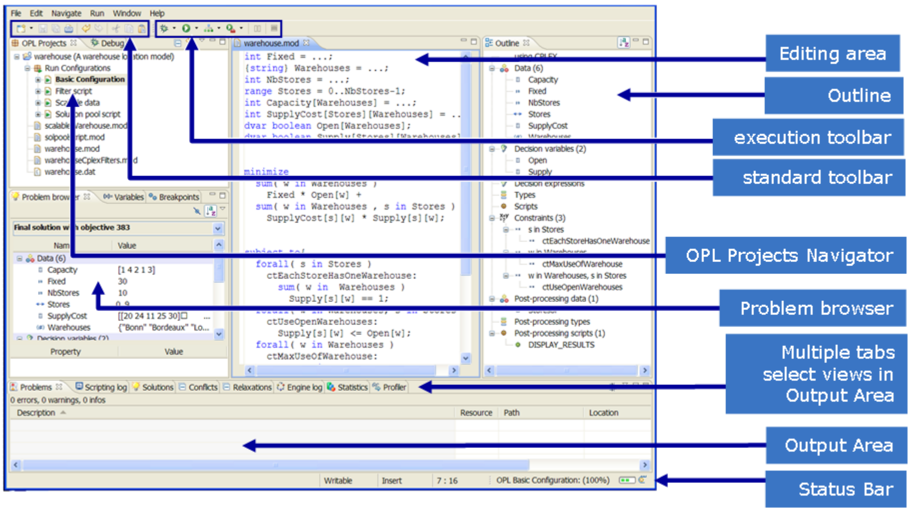
When you use the **Workbench** icon to navigate to the Main window, the Welcome window is minimized. Its elements are available from small icons in the lower righthand corner of the OPL IDE Main window.

Main window components

The Main window is organized into different **views.** A view can be an editor, a navigator, or it can provide alternative ways to visualize a project and its components.

The following diagram shows the primary views and controls of the OPL Main window. The diagram shows the Main window with a project loaded into the IDE. Tooltips appear when you move the pointer over most elements of the main window

The OPL IDE Main Window – project loaded



```
int Fixed = ...;
{string} Warehouses = ...;
int NbStores = ...;
range Stores = 0..NbStores-1;
int Capacity[Warehouses] = ...;
int SupplyCost[Stores][Warehouses] = ..
dvar boolean Open[Warehouses];

minimize
    Fixed * Open[w] +
  sum( w in Warehouses , s in Stores )
    SupplyCost[s][w] * Supply[s][w];

  forall( s in Stores )
    ctEachStoreHasOneWarehouse:
      sum( w in  Warehouses )
        Supply[s][w] == 1;
    ctUseOpenWarehouses:
      Supply[s][w] <= Open[w];
  forall( w in Warehouses )
    ctMaxUseOfWarehouse:
```

- Views in the IDE can be opened, closed, moved, resized or detached.
- The Editing area and individual table views from the Problem browser cannot be moved outside the center frame.

Lesson Title

Two of the most important areas in the OPL IDE are the:

- Editing Area
- Output Area

The Editing Area is the center of the IDE. It is where you display and work on your model, data and settings.

The Output Area contains multiple tabs that permit you to view different information produced by OPL as it processes your model:

- **Problems** displays semantic and syntax errors as you type when you write a model manually, and internal errors, such as scripting or algorithm errors, when you solve a model.
- **Scripting log** shows execution output related to the IBM® ILOG® Script **`main`** or **`execute`** or **`prepare`** blocks of the model (if applicable).
- **Solutions** displays the final solution to a model and, if applicable, any intermediate feasible solutions found.
- For infeasible CPLEX® models, **Conflicts** shows the places where you can change the data or the way filtering constraints are expressed so as to remove the incompatibilities that made the model infeasible.
- For infeasible CPLEX models, **Relaxations** shows the places that constraints can be relaxed to remove the incompatibilities that made the model infeasible.
- **Engine log** displays information from the solving engine (CPLEX or CP Optimizer) on the solving process and on the objective function.
- **Statistics** shows details of the algorithm used by the solving engine.
- **Profiler** is a tool that computes the time and memory used by each execution step and displays it as a table. You can use this information to improve the model so that it executes faster and consumes less memory. It also displays details of model extraction and engine search during the solving phase.

Instructor note
In previous versions of OPL, **Conflicts and Relaxations** were shown in the same output tab. As of OPL V6.0, they are separated.

These views and their use are explained in more detail in the OPL documentation.

Other important areas in the Main window include:

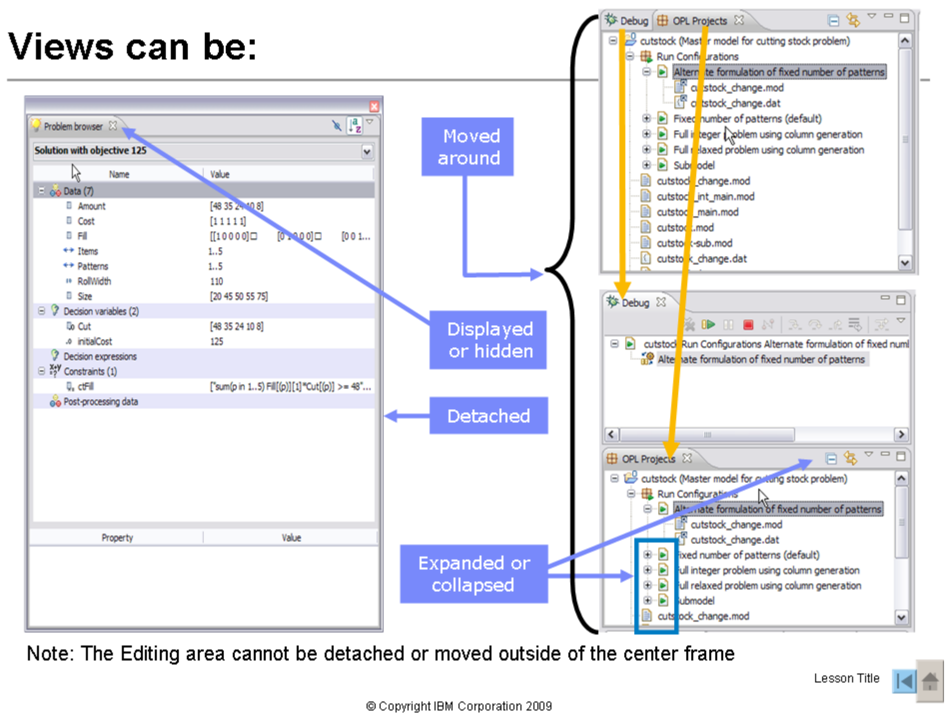
- OPL Projects Navigator – this is where you manage OPL projects and the different files that each project uses.
- Problem browser – gives you different views of critical modeling elements such as solution values, data and decision variables.
- Outline – shows the structure of the file selected in the OPL Projects Navigator.

Manipulating views

Views in the OPL IDE can be:

- opened
- closed
- moved
- resized
- detached

You can easily rearrange the IDE to suit your personal preferences or changing needs, based on what you are doing at the moment.



Views can be:

Note: The Editing area cannot be detached or moved outside of the center frame

Lesson Title

© Copyright IBM Corporation 2009

For many operations, the OPL IDE provides a variety of different ways to access commands, including:

- Main menu bar

- Context menus (available via a right-click)
- Keyboard shortcuts (such as **`<Ctl>+c`** for copying)
- Toolbar buttons

Toolbars

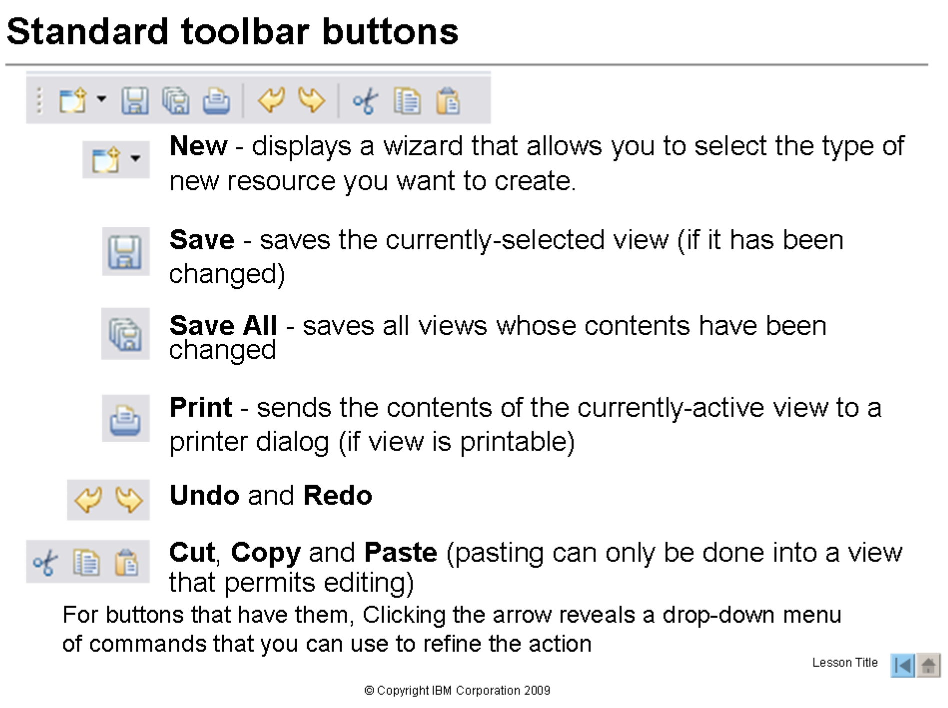
The OPL IDE provides toolbars at different levels. At the top level, two toolbars are available:

- Standard toolbar
- Execution toolbar

These toolbars are always available, although all buttons may not be active at the same time. Additional view toolbars are also available for individual views.

Many toolbar buttons are shortcuts to menu commands, but some buttons have no equivalent commands from the menu bar. When you move the pointer over a button in any toolbar, a tooltip displays a short description of it.

The standard toolbar, containing a common set of functions, is shown below with an explanation of its buttons.



Standard toolbar buttons

New - displays a wizard that allows you to select the type of new resource you want to create.

Save - saves the currently-selected view (if it has been changed)

Save All - saves all views whose contents have been changed

Print - sends the contents of the currently-active view to a printer dialog (if view is printable)

Undo and **Redo**

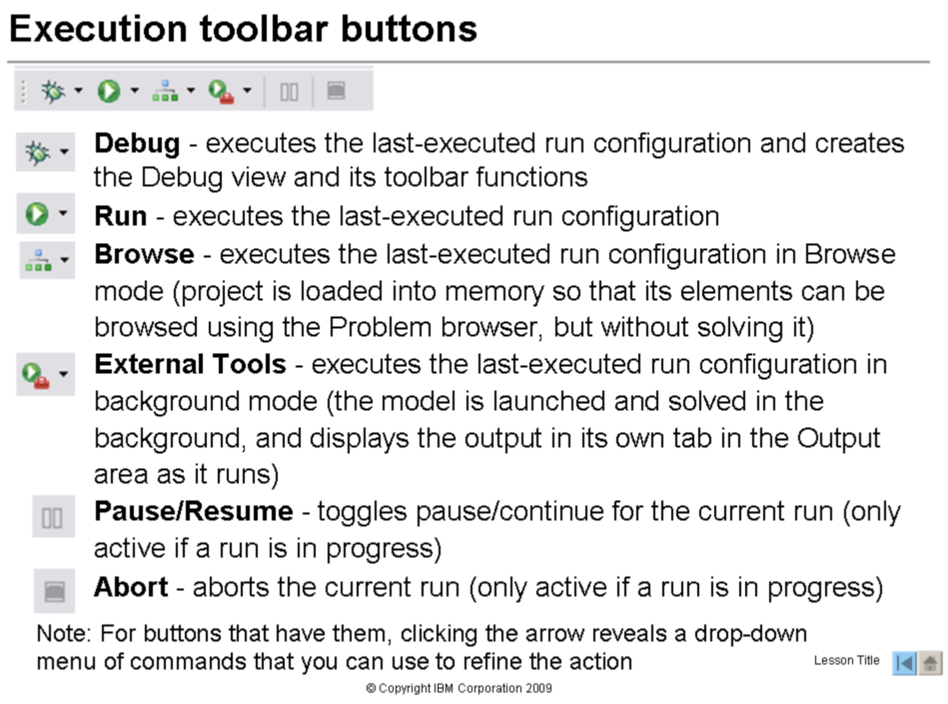
Cut, **Copy** and **Paste** (pasting can only be done into a view that permits editing)

For buttons that have them, Clicking the arrow reveals a drop-down menu of commands that you can use to refine the action

Lesson Title

© Copyright IBM Corporation 2009

The Execution toolbar provides functions related to executing projects:



Execution toolbar buttons

Debug - executes the last-executed run configuration and creates the Debug view and its toolbar functions

Run - executes the last-executed run configuration

Browse - executes the last-executed run configuration in Browse mode (project is loaded into memory so that its elements can be browsed using the Problem browser, but without solving it)

External Tools - executes the last-executed run configuration in background mode (the model is launched and solved in the background, and displays the output in its own tab in the Output area as it runs)

Pause/Resume - toggles pause/continue for the current run (only active if a run is in progress)

Abort - aborts the current run (only active if a run is in progress)

Note: For buttons that have them, clicking the arrow reveals a drop-down menu of commands that you can use to refine the action

Lesson Title

© Copyright IBM Corporation 2009

View toolbars include:

- Projects Navigator view toolbar - used to expand and collapse elements in the OPL Projects Navigator, maximize and minimize the view and customize the way the view interacts with the Editing Area.
- Problem browser view toolbar - controls how much information is displayed, sorting options and filters.
- Output Area tab views toolbars - options specific to each tab of the Output Area. Mouseover each button to see a tooltip for it.
- Debug views toolbars - used to control debugging. They are active in the three debugging views (Debug view, Variables view and Breakpoints view) only when a solution is running. They remain active as long as execution is in progress, including when a solve operation is suspended.

The OPL Preferences window

The OPL Preferences window allows you to customize many aspects of the behavior of the OPL IDE to suit your needs. Open the Preferences window by selecting **Window >Preferences** from the main menu bar.

To set options in the Preferences window:

1. In the navigation tree at the left, select the item you want to modify. Items can be expanded or collapsed using the “+” and “–” symbols before certain headings.
2. Change the settings in the fields at the right. Options for changing settings include:
    - Selecting or deselecting a check box
    - Selecting a radio button
    - Selecting items in a tree structure, list box or drop-down list
    - Filling in a text field
    - Calling up a secondary dialog box by clicking a button
    - Some of the dialogs provide links to other, related preference settings.

3. When you have finished making changes, do one of the following:
    - Click **Apply** to save your changes and continue working in preferences.
    - Click **Restore defaults** to restore default settings to all fields for the currently displayed dialog and continue working in preferences.
    - Click **OK** to save your changes and close the dialog box.
    - Click **Cancel** to keep settings unchanged and close the dialog box.

Working with projects

Learning objective
Learn about OPL projects and files, how they are created, edited and saved

Key terms
- resource
- project
- file
- folder
- workspace

Resources

A project in OPL is a type of **resource**. A resource is defined as one of the following:

- **Project:** comprised of **folders** and **files**. It maps to a directory in your computer's file system. You use a project to define the contents of a build, for version management, and to manage sharing and resource organization.
- **Folder:** similar to a folder (or subdirectory) in your computer's file system.
- **File:** similar to a file in your computer's file system

Resources are most often stored in the **workspace**, defined as the root directory in which you store and work with your resources. It can be located anywhere on the file system, but its default location is **`C:\Documents and Settings\<user_name>\Application Data\ILOG\OPL Studio IDE\6.3\`**.

The OPL Projects Navigator

The OPL Projects Navigator is where you manage your projects by creating, adding or removing resources. You can display more than one project at the same time.

The OPL Projects Navigator displays projects as tree structures with the files below them listed in alphabetical order (except for within run configurations, where the order of files is important).

Projects in the OPL Projects Navigator are persistent. Once you have imported or created projects, you can leave them in your OPL Projects Navigator. When you next launch the OPL IDE, they will be there, ready to use.

Instructor note
It may be useful to call attention, at this point, to the difference between an “open” OPL project and a “closed” one. Both appear in the workspace, but a “closed” project is not parsed at run time, and so saves CPU time and memory. This will be discussed later in the lesson, as well.

An OPL project contains:

Mandatory resources:
- The project name (and optional description in parentheses). This is the root of the tree structure.
- A default run configuration (labeled as “default” in parentheses)
- At least one model (**`.mod`**) file

Optional resources (zero, one or more):
- Additional run configurations
- Additional model (**`.mod`**) files
- Data initialization (**`.dat`**) files
- Settings (**`.ops`**) files
- If you are generating an ODM application, additional files to configure and customize the ODM application.

Creating a new project

To create an empty project:

1. In the main menu bar, select **File >New >OPL Project.** The **New Project** dialog box opens.
2. Type a name for the file in the **Project Name** field.

   Each project name has to be unique. Duplicate project names are not allowed, even if stored in different locations.

3. Use the **Browse** button (to the right of the **Project Location** field) to navigate to the directory where you want to store your project. In the navigation dialog, use the **Make New Folder** button, if necessary.

   This location is your workspace.

4. You can add a **Description** in the last field. This description appears in the **OPL Projects Navigator** next to the project name. It can be useful to distinguish between different projects with similar names and/or purposes.
5. If you want to configure custom settings for this project, check the **Create settings** box. A settings (`.ops`) file will be created for the project.

   You can create a settings file at any time, using the menu command **File >New >Settings.**

6. If you want to create a data file for this project, check the **Create Data** box. A data initialization (`.dat`) file will be created for the project.

   You can create a data initialization file at any time, using the menu command **File >New >Data.**

7. Click **Finish.** Your project is saved to your specified location and opens with an empty model file of the same name.

Importing an existing project into the OPL Projects Navigator

When you want to work on an OPL project that is not already in the OPL Projects Navigator, you need to import it.

Instructor note
This procedure applies only to OPL V6.0 projects and later. Projects from previous versions of OPL (for example, V5.x projects that use `.prj` files) need first to be migrated. Refer students to the *OPL V6.3 Migration Guide* in the documentation. If students request it, you can lead them through a migration, though it is not given in this training workbook.

Set up your import

1. From the main menu, choose **File >Import**, or right-click in the OPL Projects Navigator and choose **Import**.
2. Select **Existing Projects Into Workspace** from the submenu. The Import wizard is displayed.
3. The Import wizard offers you two choices for the location of your project. Choose one:
   - **Select root directory** to indicate the directory on your file system where the project is located. You can select it by navigating to the directory after clicking the **Browse** button.
   - **Select archive file** to indicate a compressed archive, such as a `.zip` or `.tar` file that contains the project files. Archive contents will be

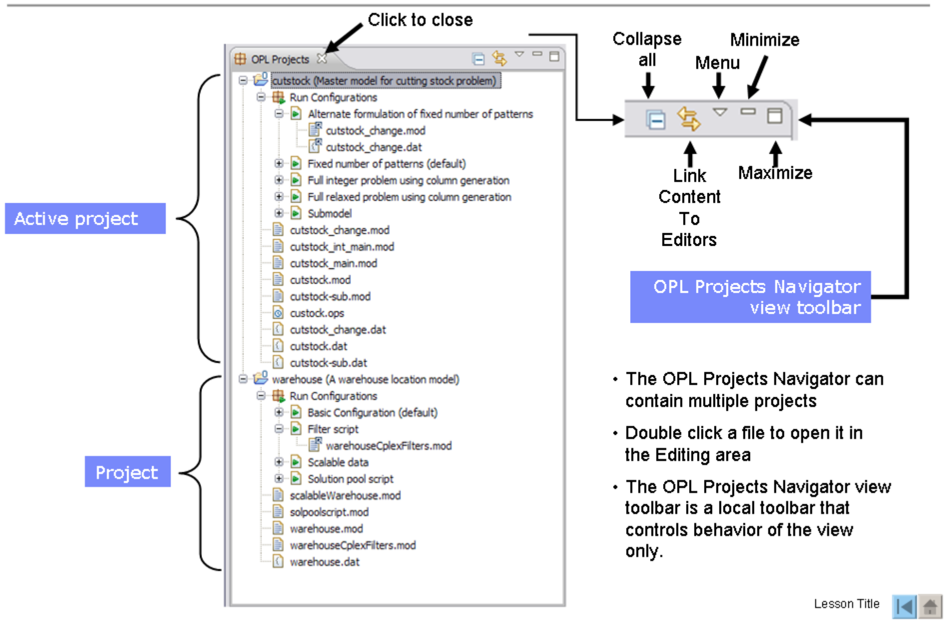
imported into the OPL Projects Navigator without the need to unarchive them. You can navigate to the archive by clicking the **Browse** button.

In the import wizard, the OPL projects in the selected directory (or archive) *that do not currently exist in your workspace* are listed in the **Projects** view.

To complete your import:

1. Check the box of each of the projects you want to import.
2. The **Copy projects into workspace** check box has the following behavior:
    - Check the box if you want to copy the project from its current location to your workspace. Any modifications you make to the project in the OPL IDE will affect only the copy in the workspace, and not the copy in the original location.
    - Leave the box unchecked if you wish to edit the original copy directly in its current location. Changes made in the OPL IDE will affect the original location, the project is not copied to the workspace.
3. Click **Finish** to complete the operation. The project has now been imported into the OPL Projects Navigator and you can work on it.

The OPL Projects Navigator



```
cutstock (Master model for cutting stock problem)
  Run Configurations
    Alternate formulation of fixed number of patterns
      cutstock_change.mod
      cutstock_change.dat
    Fixed number of patterns (default)
    Full integer problem using column generation
    Full relaxed problem using column generation
    Submodel
  cutstock_change.mod
  cutstock_int_main.mod
  cutstock_main.mod
  cutstock.mod
  cutstock-sub.mod
  custock.ops
  cutstock_change.dat
  cutstock.dat
  cutstock-sub.dat
warehouse (A warehouse location model)
  Run Configurations
    Basic Configuration (default)
    Filter script
      warehouseCplexFilters.mod
    Scalable data
    Solution pool script
  scalableWarehouse.mod
  solpoolscript.mod
  warehouse.mod
  warehouseCplexFilters.mod
  warehouse.dat
```

- The OPL Projects Navigator can contain multiple projects
- Double click a file to open it in the Editing area
- The OPL Projects Navigator view toolbar is a local toolbar that controls behavior of the view only.

Editing and saving files in projects

The Editing Area displays various contents, depending on what you select:

- The text editor appears with:
    - The contents of a model file when you double-click a **`.mod`** file in the OPL Projects Navigator
    - The contents of a data file when you double-click a **`.dat`** file in the OPL Projects Navigator
- The settings editor appears if you double-click an **`.ops`** settings file
- Specialized ODM editors appear if you have IBM ILOG ODM installed and double-click on one of the files in the **ODM Application** area of a project.

Open files for editing by double-clicking the file name in the OPL Projects Navigator. Multiple editors can be open in the Editing Area at once. You can switch back and forth between the open views by clicking the tabs of the views that are visible in the Editing Area.

You can display more than one editor at a time by grabbing the tab of a hidden window you want to display and moving it around inside the Editing Area. The different editors can be resized, but they cannot be moved out of the central frame of the IDE.

You save the file you are working on using the **File >Save** main menu selection. You can also use the equivalent toolbar button. Both menu item and toolbar button are only available when the file has been modified.

To save all files in the OPL Projects Navigator that have been modified (in all open projects), use **File >Save All** or the equivalent toolbar button.

The OPL text editor

The `.mod` and `.dat` files appear in the OPL text editor. The text editor has the following features:

- Multiple document tabs enable you to edit more than one file at the same time. You can also tile multiple documents inside the Editing Area.
- Syntax coloring: the syntax in each type of file that you can open (model, data) is colored differently, according to its type. The color scheme is customizable
- Multiple levels of Undo and Redo: you can undo and redo your modifications without any limit.
- Automatic indentation: blocks, as delimited by curly brackets {}, are automatically indented. You can increase or decrease the indentation depth by setting the Tabulation size.
- Bracket (or brace) matching: when typing], } or), the matching opening bracket is highlighted for 800 ms. In data files, < and > are also matched.
- Margin symbols: the editor has a left margin that can contain margin symbols, such as:
    - the red dot with an “X” that indicates an error
    - the blue circle that indicates a breakpoint (in debugging mode)
    - the blue circle with an arrow that indicates that OPL has stopped at the current breakpoint (in debugging mode)
    - the white arrow that indicates current location during debugging
- Reload prompt: if you modify a file with an external editor, you are prompted to reload the file as soon as the editor in the IDE regains focus.
- Customization: you can set options to customize the editor; choose **Window >Preferences** and click either **Editor** in the **General** category, or **Colors** in the **OPL** category of the navigation tree on the left of the dialog box that opens.
- Version control: The OPL IDE provides a limited form of version control called **Local History** that allows you to track and compare different versions of your files as you edit them over the life span of a project. For example, if you edit the same model file several times, all versions of the file are still available to you. You can use the **Compare With** and **Replace With** commands to compare different versions of a file or revert to previous versions of a file or its contents. These are explained fully in the documentation.

Instructor note

Many of the text editor features above are new or changed from earlier releases. Note especially the version control feature. You should make a point of reviewing these features to familiarize yourself with them before presenting this lesson. If you have students who have used older versions of OPL, you will want to point these differences out to them.

Project settings editor

You can customize the settings of each project – for example, MP, language or optimizer options – using a settings file. The settings file (with the extension `.ops`) is a separate file included in the project set. It is possible to attach multiple settings files to a project, and apply different settings files to different run configurations (see the explanation of run configurations later in this lesson). This gives you unprecedented flexibility in configuring a model and determining how it is going to solve the different problems you pass to it.

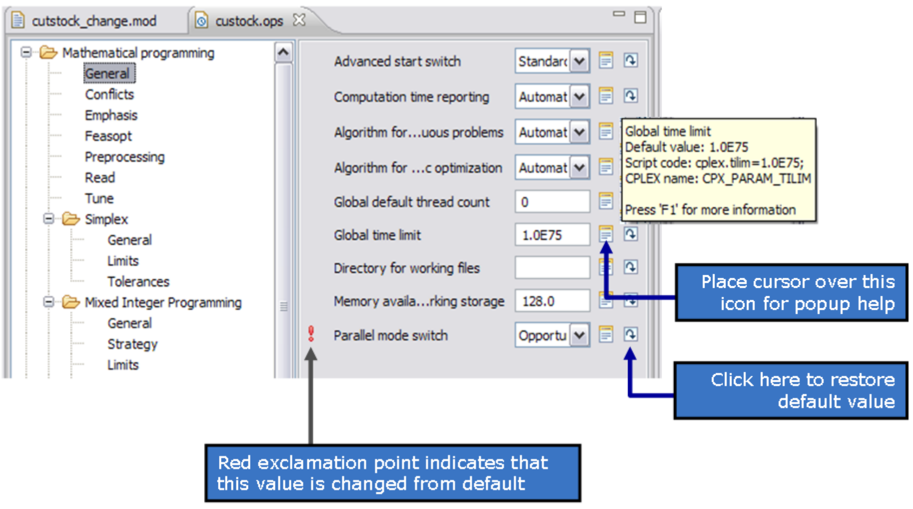
To create a new settings file, right-click anywhere in the OPL Projects Navigator, and select **New >Settings** from the context menu.

You can also create a settings file using **File >New >Settings** in the menu bar.

In both these cases, you set the parent folder and other parameters in the dialog box that opens.

To edit a settings file, double-click it in the OPL Projects Navigator. The Editing Area will show settings as in the following screen shot.

Settings file



Advanced start switch	Standard
Computation time reporting	Automat
Algorithm for…uous problems	Automat
Algorithm for …c optimization	Automat
Global default thread count	0
Global time limit	1.0E75
Directory for working files	
Memory availa…rking storage	128.0
! Parallel mode switch	Opportu

Global time limit
Default value: 1.0E75
Script code: cplex.tilim=1.0E75;
CPLEX name: CPX_PARAM_TILIM

Press 'F1' for more information

Place cursor over this icon for popup help

Click here to restore default value

Red exclamation point indicates that this value is changed from default

Lesson Title

© Copyright IBM Corporation 2009

To edit the options in the settings file:

1. In the navigation tree at the left, select the item you want to modify.
2. Change the settings in the fields at the right. Options for changing fields include:
    - Selecting items from a list box
    - Filling in a text field
    - Selecting or deselecting a check box

3.

   You will notice, to the right of each field, the symbols 📄 ↻:
   - Place the cursor over the document icon to see a popup explanation of the field and the options available.
   - Click the curved arrow to reset the field to its default value.
   - A red exclamation point to the left of a field label indicates a field that has been changed from its default value.
4. Save your changes by selecting **File >Save** from the main menu. You can also simply click the **x** in the **`.ops`** file tab to close the window. You will be prompted to save the file before the window closes.

Managing projects

Learning objective
Learn how to manage and manipulate OPL projects in the OPL Projects Navigator.

Key terms
- active project
- run configuration

You can perform the following operations on projects:

- When more than one project is in the OPL Projects Navigator, you can:
  - Set a project as the active project
  - Open or close a project
- Add a model, data or settings file to a project
- Delete a model, data or settings file from a project
- Create or remove one or more run configurations for the project
- Make a copy of a project
- Delete a project

Set the active project

The active project is the one that is currently selected in the OPL Projects Navigator. No action is required to change active projects other than clicking the project name or any of the files inside the project folder.

⚠ This does NOT mean that clicking in a project and making it the active project causes its default run configuration to be the active run configuration under the **Run** button . The behavior of this button is explained more in detail in the section on executing projects.

Close or open a project

Projects are either open or closed.

When a project is closed, it is ignored by OPL. It cannot be changed, but its resources still reside on the local file system. This can reduce build time for the project(s) that remain open.

How to close or open a project:

- To close an open project, right-click on the project name and choose **Close project** from the context menu. The plus sign next to the project name disappears, but it remains in the OPL Projects Navigator.
- To reopen the project, right-click on the project name and choose **Open project** from the context menu

Add existing files to a project

You can add existing files to a project by copying them into the selected project in the OPL Projects Navigator. There are two methods for doing this.

To add an existing file to a project – first method:

1. In the OPL Projects Navigator, select the project into which you want to import a file.

   This is an optional step, you can select the project later if you prefer, in the **Import** dialog.

2. In the main menu, select **File >Copy Files to Project**. The **Import** dialog box opens.
3. In the **Import** dialog box, indicate the source directory for the file in the **From directory** field – you can also navigate to it using the **Browse** button.
4. Select the file or files you want to add in the list that displays. You have buttons to filter the list and help select all or none of the files.

5. If you have preselected the project you are importing to, the **Into folder** field will already be filled in. You can also use the **Browse** button to select one of the project folders in your OPL Projects Navigator.
6. Select any of the **Options** you need and click **Finish**. The selected files are copied into the project and ready to be opened in the editor.

To add an existing file to a project – alternate method:

1. Open a Windows Explorer window to the directory where the file(s) you want is located.
2. Drag the file to the OPL Projects Navigator and drop the file(s) onto the project folder where you want to put them. The file(s) are copied into the project and ready to be opened in the editor.

With either method, the files are *copied* to the project, and any modifications are made only to the copy in the OPL project folder, not to the original file.

Delete a file from a project

When you remove a file from a project, you are deleting it from the project folder. This operation cannot be undone, and any changes you may have made in the file will be lost.

To delete a file from a project:

1. In the OPL Projects Navigator, right-click the file you want to remove from the project.
2. Select **Delete** from the context menu.
3. The file is removed from the project and deleted from the project folder in the file system.

When you delete a file that is referenced by a run configuration, including a run configuration in another project, all references to that file will also be deleted in all run configurations that reference it. See the section on run configurations in this lesson.

Run configurations

In the OPL IDE, a run configuration is a way of handling model, data, and settings files within a project. Basically, it is a variation of a given project for execution and testing purposes. It combines a model file and zero or more data files that differ, regarding contents and/or settings, from the original model and data of the project, while addressing the same mathematical problem. It can also contain a settings file that is specific to the run configuration.

The files listed in a run configuration are not physical files, but *references* to those files, regardless of where they are found.

You can define as many run configurations as you need within a given project.

Practically, run configurations appear as sublevels in the Projects tree.

When you create a project, the default run configuration contains only the model file. You can populate it with one or more data files, and one or more settings file.

To populate a run configuration:

1. The files you want to include must already be in your project, displayed in the OPL Projects Navigator. If not, add the file to the project first.
2. Drag and drop the file you want to add to the default run configuration (this will be called **Configuration1** unless you have renamed it).

3. You may rename the run configuration via a right-click on the run configuration name. Select **Properties** from the context menu and change the name in the dialog that opens.
4. Repeat the procedure for as many files as you wish to add to the run configuration.

If you inadvertently drop the wrong file or drop a file to the wrong place, you can at any time right-click it and choose **Delete**, then confirm. This does not remove the file from the disk.

To create a new run configuration:

1. right-click the **Run Configurations** folder in the selected project.
2. Select **New Run Configuration** from the context menu. A new run configuration with the default name **Configuration2** is added
3. Populate and rename (optional) the run configuration as outlined above.

To set a run configuration as the default:

1. right-click the run configuration that you want to set as the default run configuration.
2. Select **Set as default** from the context menu. The selected run configuration becomes the default run configuration.

Ordering files in a run configuration

When you execute a run configuration, the order of the data or settings files relative to each other is important. Since some data in a **`.dat`** file may depend on other data in a different **`.dat`** file, if the data files are in the wrong order it may cause an error at execution time. For this reason, you can reorder references to files inside a run configuration.

To set the order of files within a run configuration:

1. Right-click on the run configuration name and choose **Properties** from the context menu. A properties window appears for the run configuration.
2. Use the **Up** and **Down** buttons to rearrange the order of your settings files and data files. You can also add or remove file references in the run configuration from this window.
3. Click **OK** to close the properties window. The files in the OPL Projects Navigator are reordered.

Copy a project

You can make a copy of an existing project, and use it as the basis for a new project.

You cannot do this by simply making a copy of the original project folder on the file system and then opening it in OPL, because even if you rename the model and data files in the copy folder, the copy will have the same project name as the original.

To copy a project in the OPL Projects Navigator

1. In OPL Projects Navigator, open the project you want to copy.
2. Right-click on the project name and choose **Copy** from the context menu. You could also select the project name and press **`<Ctrl>+C`**.
3. Right-click again and choose **Paste** from the context menu, or press **`<Ctrl>+V`**.
4. In the popup window that appears, change the project name to something other than the original project name or one of the other project names currently open in OPL, and click **OK**. The new project appears in the OPL Projects Navigator.

Delete a project

If you are not currently working with a project, you can safely remove it from the OPL Projects Navigator. You have the possibility to do this without deleting it from the file system.

To remove a project from the OPL Projects Navigator:

1. Right-click on the project name and choose **Delete** from the context menu. A popup dialog appears asking whether you want to delete the project from the file system or not.
2. Two options are available to you:
    - Select **Also delete contents...** to delete the project entirely. The project will be completely deleted, and cannot later be recovered using **Undo** or the **Import >Existing Projects Into Workspace** menu command.
    - Select **Do not delete contents** (the default) to remove the project from the OPL Projects Navigator but leave it on the file system. The project is still present on the file system and can be reopened using the **Import >Existing Projects Into Workspace** menu command.

Problem browsing

Learning objective
Learn to use the **Problem browser** and **outline** views to see details of model and data structures

Key terms
- problem browser
- outline

Problem browser views

The OPL IDE provides a problem browser that lets you view model and data structures in depth. When you first load a project, the **Problem browser** window is empty of data. It fills automatically when you run a model. You can also fill it by clicking one of the options associated with the **Browse** button in the execution toolbar. In this case, however, values for decision variables and other entities that are evaluated during solution will be empty.

You can use the problem browser to:

- navigate through the model file displayed in the text editor (entities selected in the problem browser are highlighted in the text editor)
- examine the solutions to the problem and display additional views of them after executing the model
- examine the structure of the model without executing it

Refer to the documentation for details of these possibilities.

The **Problem browser** lets you visualize the properties of all the items in the model by category:

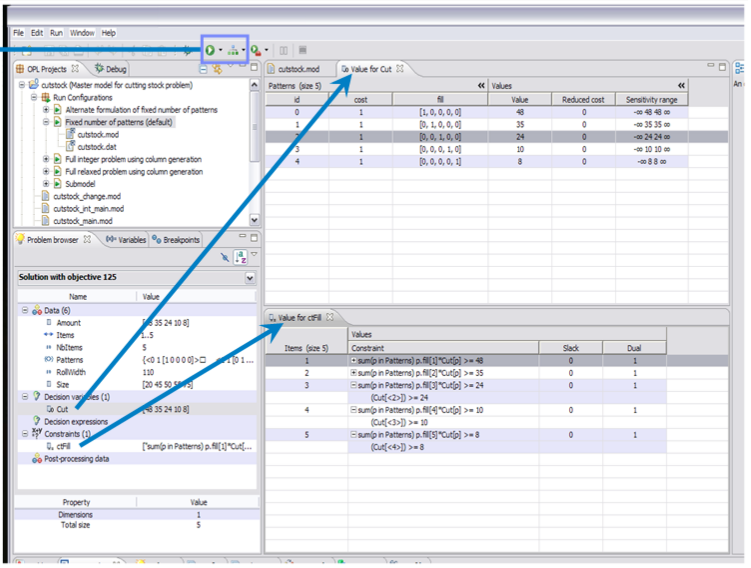
- Data structures
- Decision variables and expressions
- Constraints
- Postprocessing

The cursor moves in the source file when you select an item in the Problem browser. Item definition and values are displayed as tables in the editing area when you double-click or right-click them.

Table views in the problem browser

First, run the model or use the **Browse** button

Then, double click an item to display its table view in the editing area



File Edit Run Window Help
OPL Projects
Debug
cutstock (Master model for cutting stock problem)
Run Configurations
Alternate formulation of fixed number of patterns
Fixed number of patterns (default)
cutstock.mod
cutstock.dat
Full integer problem using column generation
Full relaxed problem using column generation
Submodel
cutstock_change.mod
cutstock_int_main.mod
cutstock_main.mod
Problem browser
Variables
Breakpoints
cutstock.mod
Value for Cut
Value for ctFill
An

Solution with objective 125

Name	Value
Data (6)	
Amount	[48 35 24 10 8]
Items	1..5
NbItems	5
Patterns	{<0 1 [1 0 0 0 0]>□ <1 1 [0 1 ...
RollWidth	110
Size	[20 45 50 55 75]
Decision variables (1)	
Cut	[48 35 24 10 8]
Decision expressions	
Constraints (1)	
ctFill	["sum(p in Patterns) p.fill[1]*Cut[...
Post-processing data	

Property	Value
Dimensions	1
Total size	5

Patterns (size 5)			Values		
id	cost	fill	Value	Reduced cost	Sensitivity range
0	1	[1, 0, 0, 0, 0]	48	0	-∞ 48 48 ∞
1	1	[0, 1, 0, 0, 0]	35	0	-∞ 35 35 ∞
2	1	[0, 0, 1, 0, 0]	24	0	-∞ 24 24 ∞
3	1	[0, 0, 0, 1, 0]	10	0	-∞ 10 10 ∞
4	1	[0, 0, 0, 0, 1]	8	0	-∞ 8 8 ∞

	Values		
Items (size 5)	Constraint	Slack	Dual
1	sum(p in Patterns) p.fill[1]*Cut[p] >= 48	0	1
2	sum(p in Patterns) p.fill[2]*Cut[p] >= 35	0	1
3	sum(p in Patterns) p.fill[3]*Cut[p] >= 24	0	1
	(Cut[<2>]) >= 24		
4	sum(p in Patterns) p.fill[4]*Cut[p] >= 10	0	1
	(Cut[<3>]) >= 10		
5	sum(p in Patterns) p.fill[5]*Cut[p] >= 8	0	1
	(Cut[<4>]) >= 8		

Lesson Title

Depending on the type of problem and the structures in it, one or more parameters are shown in the problem browser table views:

Parameters for decision variables:

- Value
- Reduced cost
- Sensitivity range

Parameters for constraints:

- Slack
- Dual values

The Outline

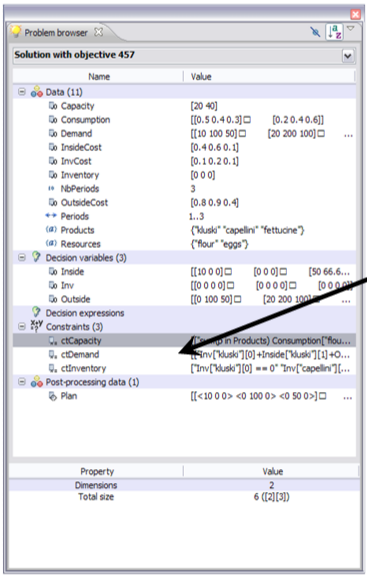
The **outline** window shows two different views, depending on what is open in the text editor:

- When a **.mod** file is open in the workspace, the **Model outline** window is displayed.
- When an **.ops** settings file is open in the workspace, the **Settings outline** window is displayed.
- The **outline** window is blank when a **.dat** file is opened.

The **Model outline** window displays the syntax of the model as a tree structure. For each type of element in the model, the element type and number of elements in the model of that type are indicated. To display the outline of a model, just double-click the **.mod** file in the project tree.

The following illustration compares the **Problem browser** and **Model outline** views for the same problem:

Comparison: problem browser & model outline



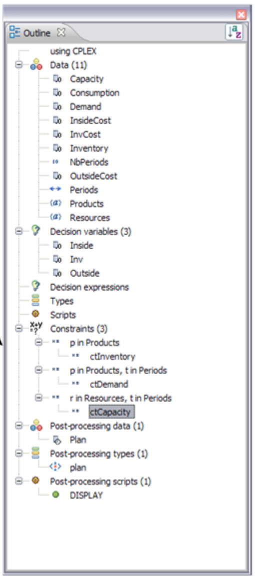
Problem browser

Solution with objective 457

Name	Value
Data (11)	
Capacity	[20 40]
Consumption	[[0.5 0.4 0.3]□ [0.2 0.4 0.6]]
Demand	[[10 100 50]□ [20 200 100]□ …
InsideCost	[0.4 0.6 0.1]
InvCost	[0.1 0.2 0.1]
Inventory	[0 0 0]
NbPeriods	3
OutsideCost	[0.8 0.9 0.4]
Periods	1..3
Products	{"kluski" "capellini" "fettucine"}
Resources	{"flour" "eggs"}
Decision variables (3)	
Inside	[[10 0 0]□ [0 0 0]□ [50 66.6…
Inv	[[0 0 0 0]□ [0 0 0 0]□ [0 0 0 0]]
Outside	[[0 100 50]□ [20 200 100]□ …
Decision expressions	
Constraints (3)	
ctCapacity	[["sum(p in Products) Consumption["flou…
ctDemand	[["Inv["kluski"][0]+Inside["kluski"][1]+O…
ctInventory	["Inv["kluski"][0] == 0" "Inv["capellini"][…
Post-processing data (1)	
Plan	[[<10 0 0> <0 100 0> <0 50 0>]□ …

Property	Value
Dimensions	2
Total size	6 ([2][3])

Note the difference in how constraints are shown



Outline

```
using CPLEX
Data (11)
    Capacity
    Consumption
    Demand
    InsideCost
    InvCost
    Inventory
    NbPeriods
    OutsideCost
    Periods
    Products
    Resources
Decision variables (3)
    Inside
    Inv
    Outside
Decision expressions
Types
Scripts
Constraints (3)
    p in Products
        ctInventory
    p in Products, t in Periods
        ctDemand
    r in Resources, t in Periods
        ctCapacity
Post-processing data (1)
    Plan
Post-processing types (1)
    plan
Post-processing scripts (1)
    DISPLAY
```

Lesson Title

When editing a settings file, the **Settings outline** window is automatically displayed. It shows, in outline format, a summary of all parameters whose values have been changed from the default.

Solving and debugging

Learning objective
Learn about the IDE tools used to run and debug solves of OPL models

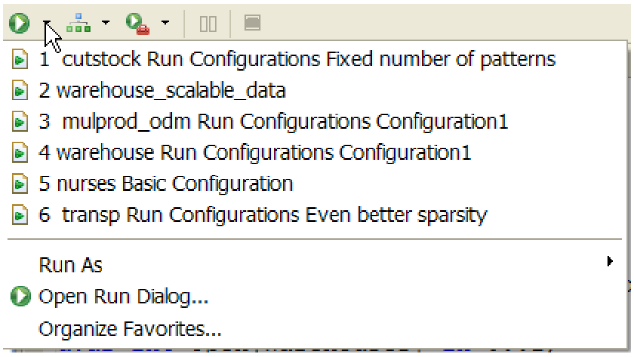
Key terms
- execution
- errors
- infeasibility
- breakpoint

The OPL IDE relies on the CPLEX® and CP Optimizer engines to solve problems. It displays results from the engines in the various **Output** tabs. When errors occur, the OPL IDE allows you to insert **breakpoints** to help you examine the state of your model during different phases of solution and correct the error.

The Run button

The behavior of the **Run** button in the execution toolbar depends on your "run history."

- If you have just launched OPL and no models been run yet, clicking the **Run** button may produce an error message.
- As runs are executed, they are added to a numbered list that is visible by clicking the arrow to the right of the **Run** button, as shown in the following screen shot:



1 cutstock Run Configurations Fixed number of patterns
2 warehouse_scalable_data
3 mulprod_odm Run Configurations Configuration1
4 warehouse Run Configurations Configuration1
5 nurses Basic Configuration
6 transp Run Configurations Even better sparsity

Run As
Open Run Dialog...
Organize Favorites...

- Once this list is populated, clicking the **Run** button launches *the most recently launched run configuration* in the list, no matter what project is selected in the OPL Projects Navigator.
- The default behavior of the **Run** button is configurable. See the documentation for details.

It should be obvious that it is not possible to simply select a project in the OPL Projects Navigator and launch its default run configuration by a single click on the **Run** button.

For this reason, many OPL developers prefer the right-click context menus to launch their OPL solves.

The Run context menus

You can use context menus to launch solves of your projects directly from the OPL Projects Navigator.

To run projects from the OPL Projects Navigator – first method:

1. right-click on the project folder, or on any resource in the project folder *except* an individual run configuration. A context menu appears.
2. Select **Run** and then choose one of the two options presented:

- Select **Default Run Configuration** – this option runs the run configuration that is currently set as the default for this project.
- Select one of the run configurations shown in the list. All run configurations for the project are shown here, and you can choose which one you want to launch, whether it is the current default run configuration or not.

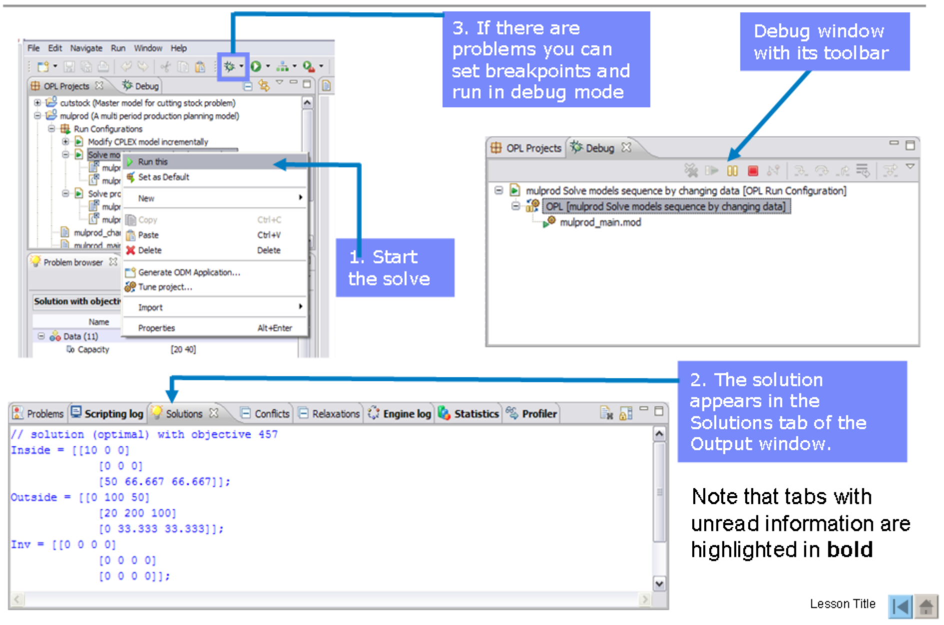
To run projects from the OPL Projects Navigator – alternate method:

1. right-click the run configuration you wish to use for the solve. A context menu appears.
2. Select **Run this** from the context menu. You can only run the selected run configuration.

Finding a solution

The following diagram shows the steps to execute a model.

To find a solution



3. If there are problems you can set breakpoints and run in debug mode
Debug window with its toolbar
1. Start the solve
2. The solution appears in the Solutions tab of the Output window.
Note that tabs with unread information are highlighted in **bold**
Run this
Set as Default
Problems
Scripting log
Solutions
Conflicts
Relaxations
Engine log
Statistics
Profiler
Lesson Title

```
// solution (optimal) with objective 457
Inside = [[10 0 0]
             [0 0 0]
             [50 66.667 66.667]];
Outside = [[0 100 50]
             [20 200 100]
             [0 33.333 33.333]];
Inv = [[0 0 0 0]
             [0 0 0 0]
             [0 0 0 0]];
```

Results are displayed in the different tabs of the Output Area, as described earlier in this lesson.

Errors

There are 4 reasons why an execution does not produce a solution:

- Syntax errors - for example: you forget a ';' after an instruction
- Semantic errors - for example:
  - A variable name is misspelled
  - Initialization type does not match declaration
- Run time errors - for example: an array initialization list is too long
- Infeasibility - for example: two constraints are incompatible

In the case of syntax or semantic errors, the text editor window highlights the source code line where the error is found dynamically as you edit.

Run time errors are highlighted after running a model. In the **Problems** tab of the **Output** window, details are given about the error(s) found. Once corrected, you can immediately run the model again.

In the case of infeasibility, the **Conflicts** tab identifies the constraints in the conflict and the **Relaxations** tab and proposes a relaxation that will produce a solution.

Debugging

The IBM ILOG OPL IDE provides three different views for debugging:

- **Debug** view
- **Variable** view
- **Breakpoint** view

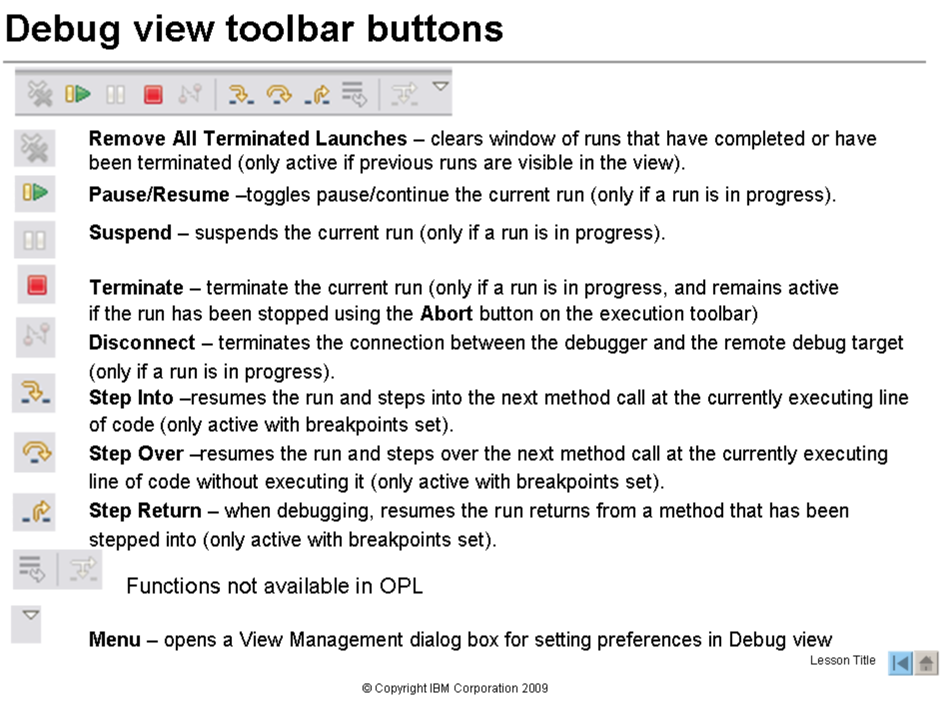
When your model contains IBM ILOG Script **`main`** or **`execute`** blocks, you can set breakpoints inside them by double-clicking the lefthand margin in the gray area. A blue dot appears to indicate the location of the breakpoint.

When you insert a break point in IBM ILOG Script **`main`** or **`execute`** block, you can use the **Debug** button on the execute toolbar to stop execution at the indicated breakpoint.

You can then inspect the call stack in two views:

- The Debug view shows the nested function calls. Each function called has information in a stack frame. You can expand and collapse elements by clicking the **+** or **–** signs.
- The Variable view shows the content of the selected call frame.

The following diagram shows the Debug view toolbar buttons and functions. This toolbar is local to the Debug view.



Debug view toolbar buttons

Remove All Terminated Launches – clears window of runs that have completed or have been terminated (only active if previous runs are visible in the view).

Pause/Resume –toggles pause/continue the current run (only if a run is in progress).

Suspend – suspends the current run (only if a run is in progress).

Terminate – terminate the current run (only if a run is in progress, and remains active if the run has been stopped using the **Abort** button on the execution toolbar)

Disconnect – terminates the connection between the debugger and the remote debug target (only if a run is in progress).

Step Into –resumes the run and steps into the next method call at the currently executing line of code (only active with breakpoints set).

Step Over –resumes the run and steps over the next method call at the currently executing line of code without executing it (only active with breakpoints set).

Step Return – when debugging, resumes the run returns from a method that has been stepped into (only active with breakpoints set).

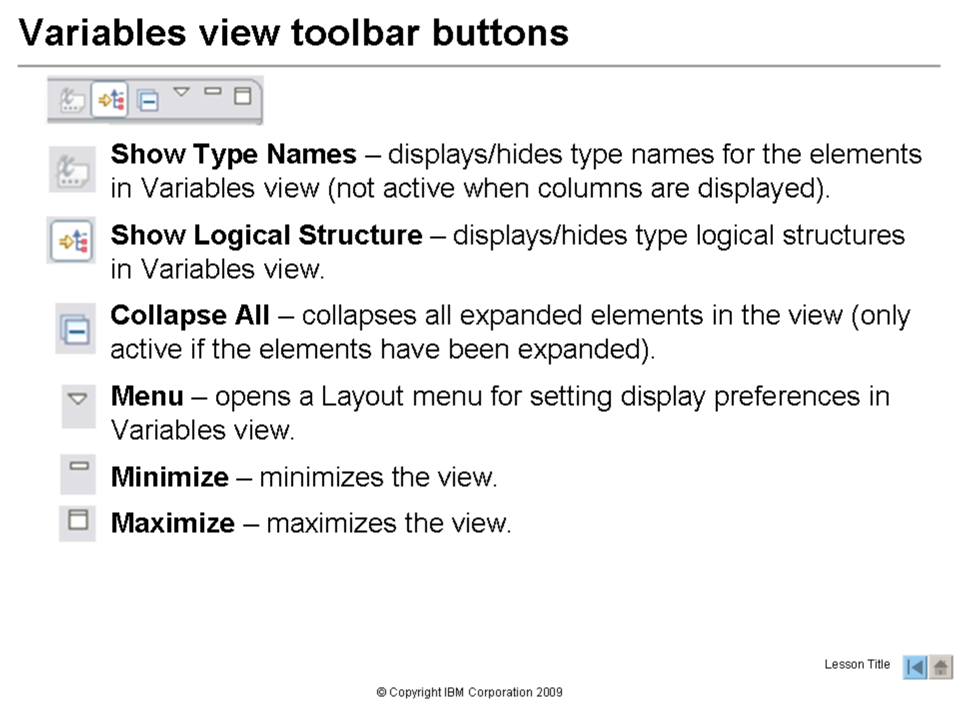
Functions not available in OPL

Menu – opens a View Management dialog box for setting preferences in Debug view

Lesson Title

© Copyright IBM Corporation 2009

The following diagram shows the Variable view toolbar buttons and functions. This toolbar is local to the Variable view.



Variables view toolbar buttons

Show Type Names – displays/hides type names for the elements in Variables view (not active when columns are displayed).

Show Logical Structure – displays/hides type logical structures in Variables view.

Collapse All – collapses all expanded elements in the view (only active if the elements have been expanded).

Menu – opens a Layout menu for setting display preferences in Variables view.

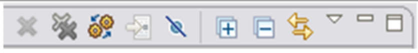
Minimize – minimizes the view.

Maximize – maximizes the view.

Lesson Title

The following diagram shows the Breakpoints view toolbar buttons and functions. This toolbar is local to the Variable view.

Breakpoints view toolbar buttons



Remove Selected Breakpoints – delete the currently-selected breakpoints from the view.

Remove All Breakpoints – delete all current breakpoints from the view.

Show Breakpoints Supported By Selected Target – displays/hides breakpoints supported/not supported by the currently-selected debug target.

Function not available in OPL

Skip All Breakpoints – mark all breakpoints in the current view as skipped.

Expand All – expands all collapsed elements in the view (only if the elements have been collapsed).

Collapse All – collapses all expanded elements in the view (only active if the elements have been expanded).

Link With Debug View – updates Breakpoints view with information from Debug view.

Menu – opens a menu for setting preferences in Breakpoints view.

Minimize – minimizes the view.

Maximize – maximizes the view.

Lesson Title

Summary

Review

In this lesson, you learned about the OPL IDE user interface and its functions.

Conclusion

In this training, you have seen that OPL is a prototyping language for optimization applications with:

- Support for:
    - Linear programming
    - Integer and Mixed Integer programming
    - Quadratic and Quadratically Constrained Programs
    - Constraint programming, including detailed scheduling
- Ability to define and control the search procedures
- Rich and efficient data structures

The OPL IDE is a complete development environment including:

- Project management
- Dynamic display of data
- Several stepping debug modes
- Full data browsing capacity

In addition, the OPL IDE provides a rich set of external links to provide a variety of deployment options, including:

- IBM® ILOG® ODM, an application-building environment integrated with OPL that lets business users to directly change scenarios and perform "what-if" analysis in a user-friendly, interactive environment.
- The OPL Interfaces, which provide APIs for:
    - C++
    - Java™
    - Visual Basic .Net
    - Microsoft® Office applications
- Direct communication with leading databases and spreadsheets to read data and store solutions.

You have successfully used OPL in a simulation of some of your everyday business practices.

The next step is to look at the documentation that accompanies OPL (if you have not already). The documentation can be found within the standard help system of the graphical environment provided in the OPL IDE. To access the Online Help, click **Help** on the **Help** menu. Contextual help in the OPL IDE is available via the **`F1`** key.

For more information:

http://www.ilog.com

Technical support pages:

Open service requests:
http://www.ibm.com/support/electronic/uprtransition.wss?category=2locale=en_us.

This is a tool to help clients find the right place to open any problem, hardware or software, in any country where IBM does business. This is the starting place when it is not evident where to go to open a service request.

Service Request (SR): http://www.ibm.com/software/support/probsub.html

This page offers Passport Advantage clients for distributed platforms online problem management to open, edit and track open and closed PMRs by customer number.

You can find information about assistance for SR at
http://www.ibm.com/software/support/help-contactus.html