Java and CPLEX

Paul Bouman

Overview

- The JVM and Native Code
- Libraries and Packages
- Linear Programming and the CPLEX library
- Example: Precedence Constrained Knapsack Problem
- Model Class
- Solving and Updating the Model
- Final Hints and Tips

The JVM and Native Code

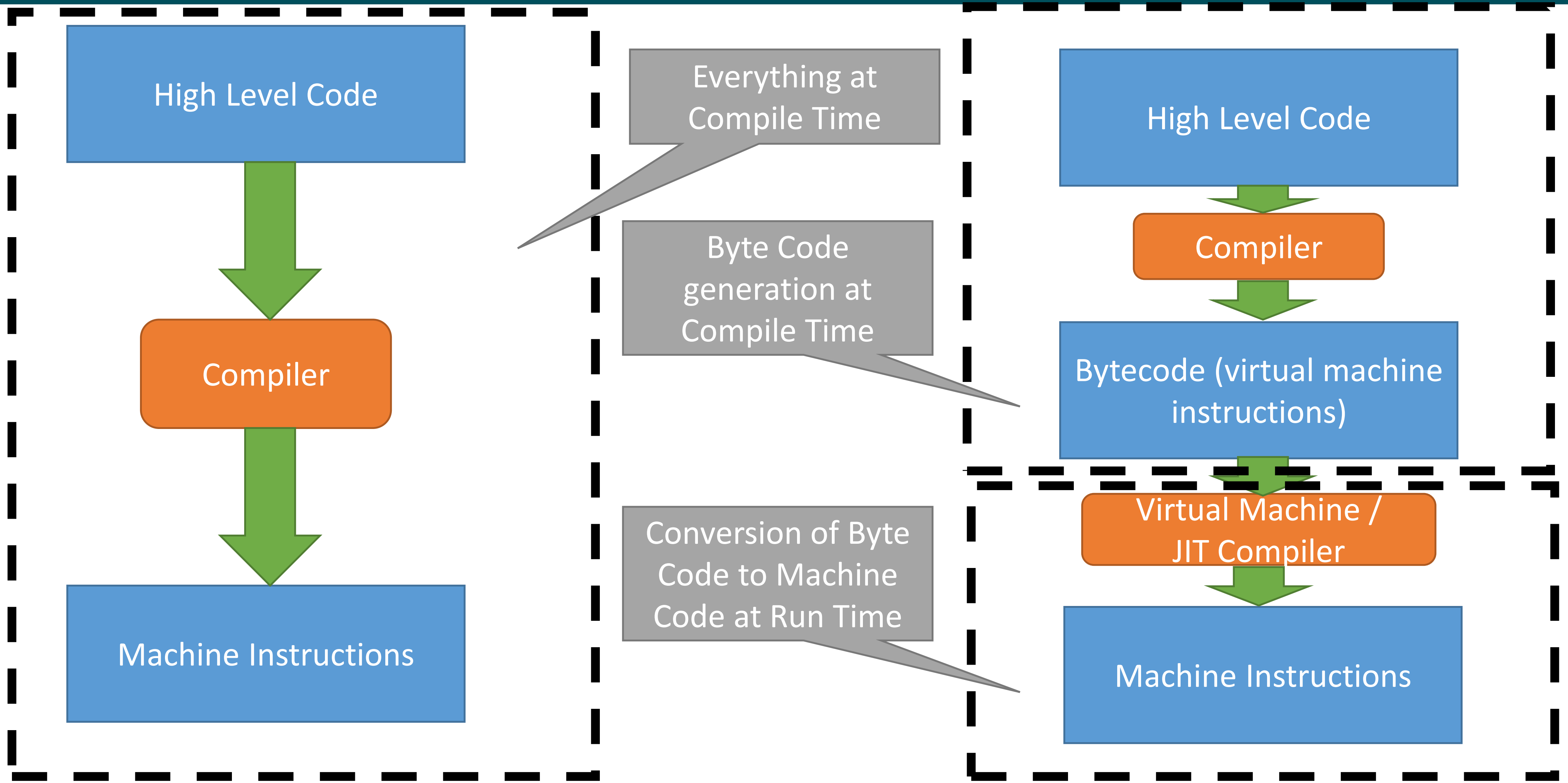
Compilers

- Internally, computers work with a fixed set of simple instructions.
- Writing these instructions directly is called **low level programming**.
  - No features such as loops, functions, objects, etc. Only (conditional) jumps, and elementary data types.
- Since this is error-prone and cumbersome, most programmers use **high level programming languages** (such as Java)
- As a result we need a **compiler** which translates a high-level program into machine instructions to be executed by the computer.
- Traditional compilers (e.g. for C and C++) convert high-level code into machine instructions directly.

Compilers – Virtual Machines

- The Java compiler produces **bytecode**, which are best described as machine instructions for a fictional machine. Regular computers still don’t understand these instructions!
- This bytecode is translated into machine code by the **Java Virtual Machine (JVM)**.
  - Early JVM implementations were inefficient, but over 20 years of R&D have resulted in a very optimized and efficient JVM which is often competitive with native implementations in terms of speed.
- **Advantage**: you compile once and can run your program on any type of hardware or OS which has a JVM implementation
- **Disadvantage**: you lose some of the nitty-gritty control that can be important when you want to exploit specific strengths of particular hardware

Compilers – Traditional Languages vs VM Languages



Native Code

- Some specialistic software has been in development for decades and is very optimized.
  - LAPACK (Linear Algebra Package) is written in FORTRAN and used by MATLAB, numpy, for linear algebra computations
  - CPLEX library for optimization
  - Many others…
- The Java Native Interface (JNI) allows us to make use of very efficient libraries that were not written in Java.

Libraries and Packages

Libraries

“If I have seen further it is by **standing on the shoulders of giants**”

Sir Isaac Newton (1676)

Libraries

- In software development, Newton’s famous quote is just as relevant as in science in general.
- The Collections framework is a good example of this: instead of having to program complicated data structures and sorting algorithms by yourself, you can use them as building blocks for your programs.
- For many specialistic topics, there is no framework included with the standard Java distribution, but many programmers provide libraries for specific tasks.
    - Think about matrix operations, solving mathematical optimization problems, performing statistical tests, etc.

Java - Packages

- A package contains a bunch of classes or interfaces that “naturally” belong together.
- Often they are some URL reversed, like `org.apache.commons`, `java.util`, `java.io`
- Classes and interfaces from different packages than “the current one” need to be imported (it is often best to let Eclipse handle this).
- When using a library Java will add a number of packages, classes and interfaces for us to use, just like the ones we can already use, such as `ArrayList`

Java - Libraries

- Java libraries come in two flavours:
- **Pure Java Libraries** where everything in written in Java and compiled to bytecode. These have the advantage that can run everywhere and are relatively easy to include to your project (you add the library to the classpath and you are done)
- **Native Java Libraries** where you have both a **Java** component you have to add to the classpath, as well as a **native library** (Windows: .dll, Mac/Linux: .so) which has to be added to the native library path of the **JVM**. You also lose portability.
- Today we discuss the IBM ILOG CPLEX library as an example of a native library. Using a **pure Java library** typically requires less effort.

Java – Importing a Library Into Eclipse

- Typical steps
  1. Create a directory lib in your project and put the relevant files there
  2. Go to the project properties, which can be access by right clicking on your project folder or via the Project menu in the menu bar.
  3. Go to the Java Build Path option
  4. Press add jars and select a .jar file to the **classpath (see next slide)**
  5. If it is a native library: add the path of the native library (only directory)

- Watch the YouTube video if you need help.

Java – Classpath / Modulepath

- Eclipse Projects that are configured with a Java version up to 8 only have the **classpath** on which you libraries must be loaded.
- Project configured with Java version 9 or later have a **classpath** and a **module path.**
- Consider to create a Java 8 project (Use JavaSE-1.8 as the execution environment)
- Easiest solution for most small scale Java 9+ projects:
  - Don't create a `module-info.java` file and add the libraries to the classpath.
  - Delete `module-info.java` if you created it.
- For CPLEX: afaik no module-based version. If you really want to use the module system (not recommended):
  - Add cplex.jar to the module path
  - Add requires cplex; to your module-info.java file

Java – Classpath / Modulepath

- Eclipse Projects that are configured with a Java version up to 8 only have the **classpath** on which you libraries must be loaded.
- Project configured with Java version 9 or later have a **classpath** and a **module path.**
- Consider to create a Java 8 project (Use JavaSE-1.8 as the execution environment)
- Easiest solution for most small scale Java 9+ projects:
  - Don't create a `module-info.java` file and add the libraries to the classpath.
  - Delete `module-info.java` if you created it.
- For CPLEX: afaik no module-based version. If you really want to use the module system (not recommended):
  - Add cplex.jar to the module path
  - Add requires cplex; to your module-info.java file

Design Choices – Obtaining Objects

- When working with a library, we will probably want to obtain some objects
  - Unless everything in the library is `static`, but that is not very common.
- The most likely ways are:
  - Calling constructors, eg. `new ArrayList<Integer>();`
  - Calling static methods, eg. `BigInteger.valueOf(12);`
  - Being created by another object, eg. `list.iterator();`
- To figure out which style or styles are being used by your library of choice, you should **Read the Friendly Manual**
  - Sometimes you have a good step-by-step tutorial that gets you started
  - Sometimes there are some examples that you can study.
  - In other case you should search through API documentation and/or JavaDocs to figure out how to create and use objects.
  - It never hurts to experiment a bit!

Linear Programming and the CPLEX library

Linear Programming

Different types of linear programs exists.

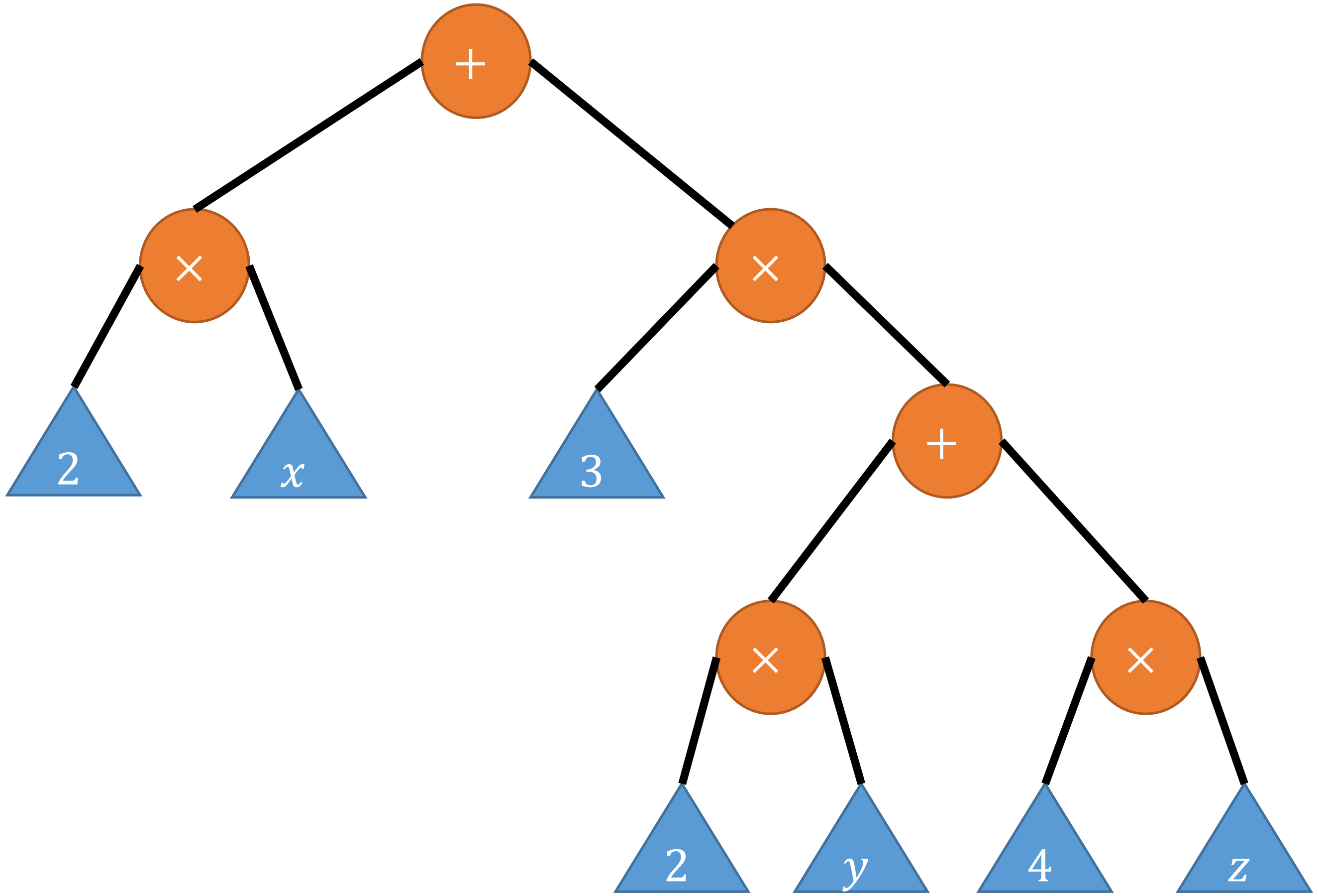
Linear Program *(sometimes LP-relaxation of)*	Minimize or Maximize cx Subject to $Ax \leq b$ $x \in \mathbb{R}^d$	Easy to solve: Simplex Method, Interior Point Methods
Integer Linear Program	Minimize or Maximize qy Subject to $Dy \leq b$ $y \in \mathbb{N}^k$	Hard to solve: use Linear Programming Relaxation + Branching Algorithm
Mixed Integer Linear Program	Minimize or Maximize $cx + qy$ Subject to $Ax + Dy \leq b$ $x \in \mathbb{R}^d, y \in \mathbb{N}^k$	Same as Integer Linear Program

The Grammar of Formulas

- A formula usually creates a relationship between mathematical expressions.
- The building blocks of mathematical expressions are:
  - Constants: $2, 8.6, \pi$
  - Variables: x_i, y_j
  - Operators: $\times, +, -$, etc.
  - Parenthesis (to avoid ambiguity): ( )
- An **mathematical expression** is either:
  1. A constant **or** a variable
  2. An **mathematical expression** surrounded by parentheses
  3. Two **mathematical expressions** connected by an operator

The Grammar of Formulas

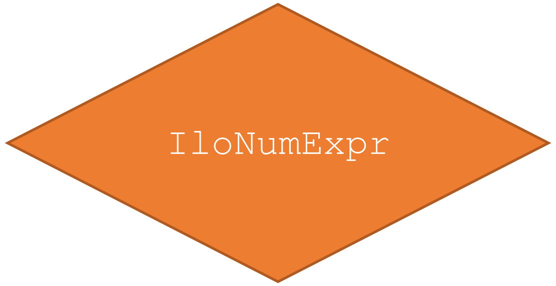
- Example: $2x + 3 \cdot (\ 2y + 4z\ )$



The Grammar of Formulas

- Mathematical expressions can be constructed by combining smaller expressions recursively, in a tree structure
- A formula is often a relationship between two expressions:
  - $expr_1 \leq expr_2$
  - $expr_1 \geq expr_2$
  - $expr_1 = expr_2$
- The CPLEX library offers “Concert Technology” which allows us to build models by means of larger and larger expressions, starting with constants and variables.
- These expressions can be used to add **constraints** and optimization **objectives** to your mathematical model.

CPLEX Class Hierarchy



CPLEX Class Hierarchy

Interface IloNumExpr

All Known Subinterfaces:

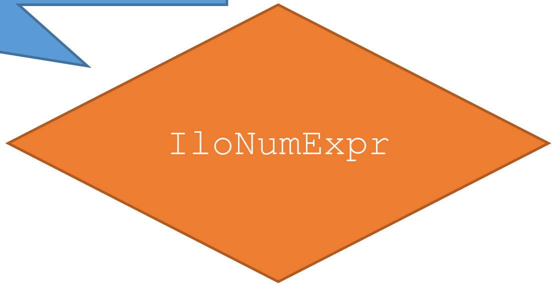
IloAnd, IloConstraint, IloIntExpr, IloIntVar, IloLinearIntExpr, IloLinearNumExpr, IloLPMatrix, IloLQIntExpr, IloLQNumExpr, IloNumVar, IloNumVarBound, IloOr, IloQuadIntExpr, IloQuadNumExpr, IloRange, IloSemiContVar, IloSOS1, IloSOS2

`public interface` **`IloNumExpr`**

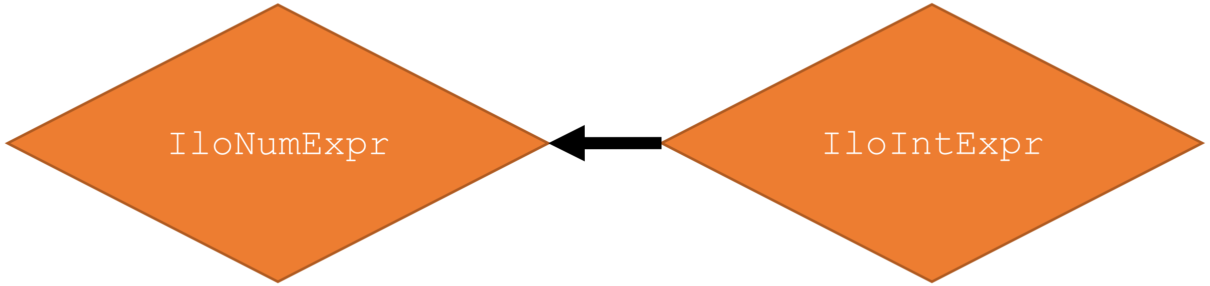
This is the public basic interface for all numerical expressions. Numerical expressions are represented by objects implementing this interface. They are constructed using the expression operator functions defined in the interface `IloModel` or one of its extensions.

Concert Technology distinguishes integer expressions that are built solely from integer variables and use only integer values. Integer expressions are represented by the interface `IloIntExpr`, an extension of `IloNumExpr`. Integer expressions can be used wherever general expressions of type `IloNumExpr` are expected.

Variables defined by the interface `IloNumVar` or `IloIntVar` are also extensions of `IloNumExpr`. Therefore, variables can be used wherever general expressions are expected.



CPLEX Class Hierarchy



CPLEX Class Hierarchy

Interface IloIntExpr

All Superinterfaces:

IloNumExpr

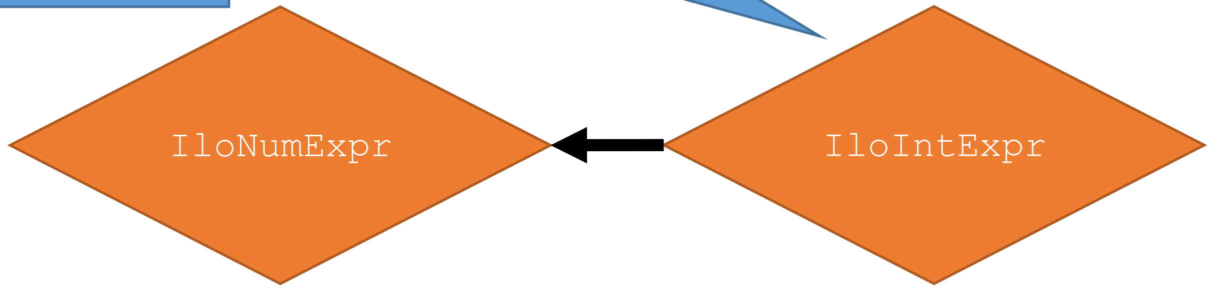
All Known Subinterfaces:

IloAnd, IloConstraint, IloIntVar, IloLinearIntExpr, IloLPMatrix, IloLQIntExpr, IloNumVarBound, IloOr, IloQuadIntExpr, IloRange, IloSOS1, IloSOS2

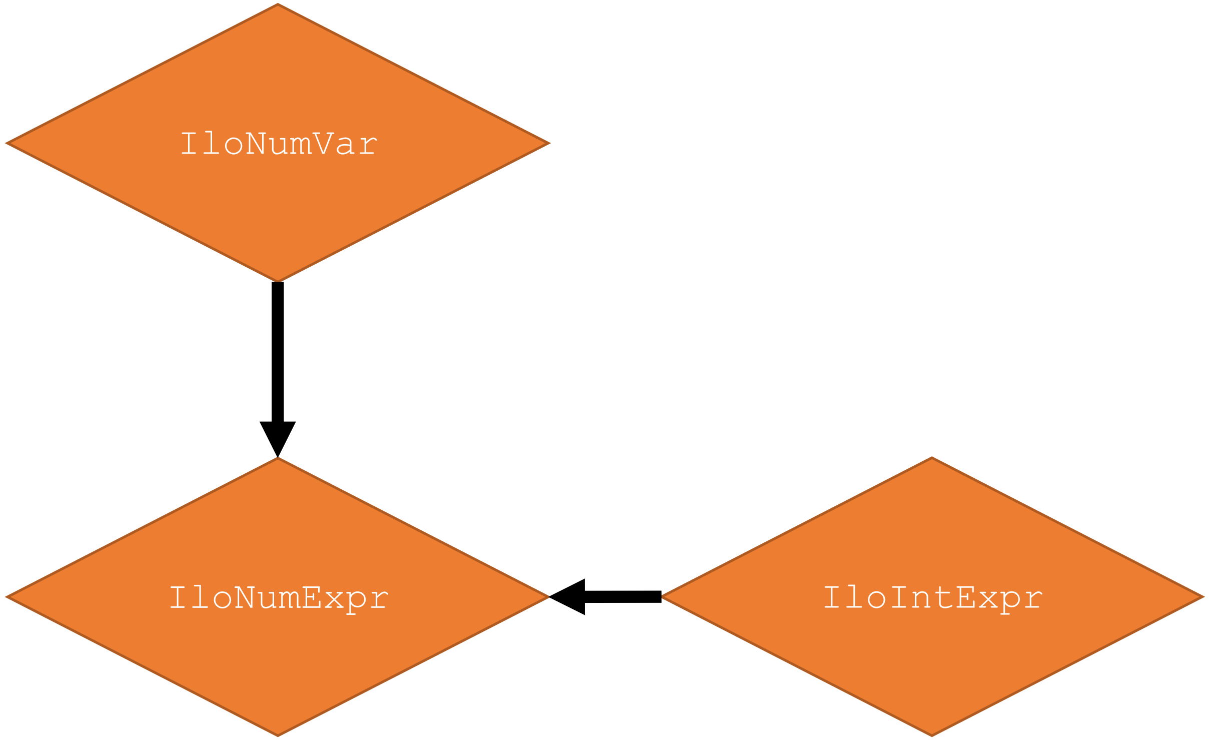
```
public interface IloIntExpr
extends IloNumExpr
```

This is the basic public interface for integer expressions. Integer expressions are represented using objects of type `IloIntExpr`. They are guaranteed to contain only variables of type `integer` and perform integer arithmetic. Integer expressions are created using integer variables and values with the numerical operations provided in `IloModeler`.

Integer expressions and general expressions can be mixed. This is achieved by defining the interface `IloIntExpr` as an extension of `IloNumExpr`. However, when an integer expression is used as an instance of `IloNumExpr`, the compile-time information is lost. For some optimizers, this will incur a runtime overhead because the type information needs to be regained at run time. This is documented for the optimizers where it is relevant.



CPLEX Class Hierarchy



CPLEX Class Hierarchy

Interface IloNumVar

All Superinterfaces:

IloAddable, IloNumExpr

All Known Subinterfaces:

IloIntVar, IloSemiContVar

```
public interface IloNumVar
extends IloNumExpr, IloAddable
```

This interface defines the API for numerical variables of any type.

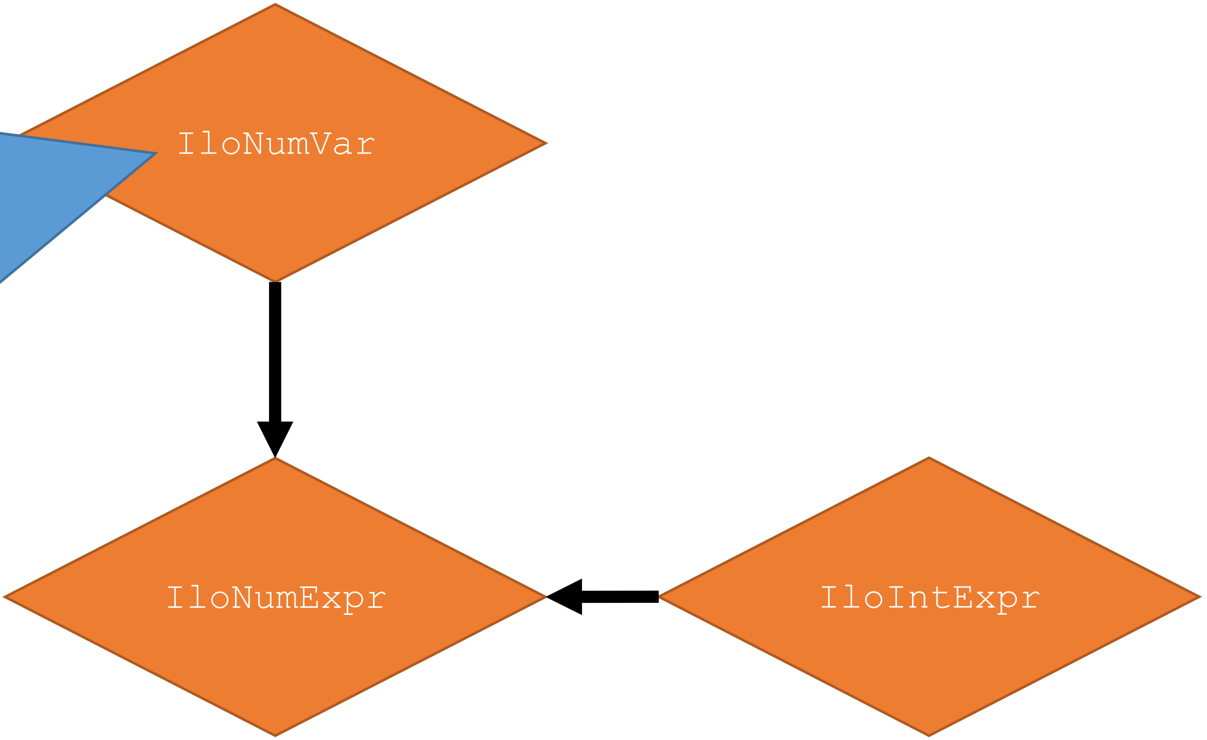
Objects implementing this interface are used to represent modeling variables in IBM® ILOG® Concert Technology. A modeling variable is characterized by its lower and upper bounds as well as by its type. Possible types are:

- `IloNumVarType.Float`
- `IloNumVarType.Int`
- `IloNumVarType.Bool`

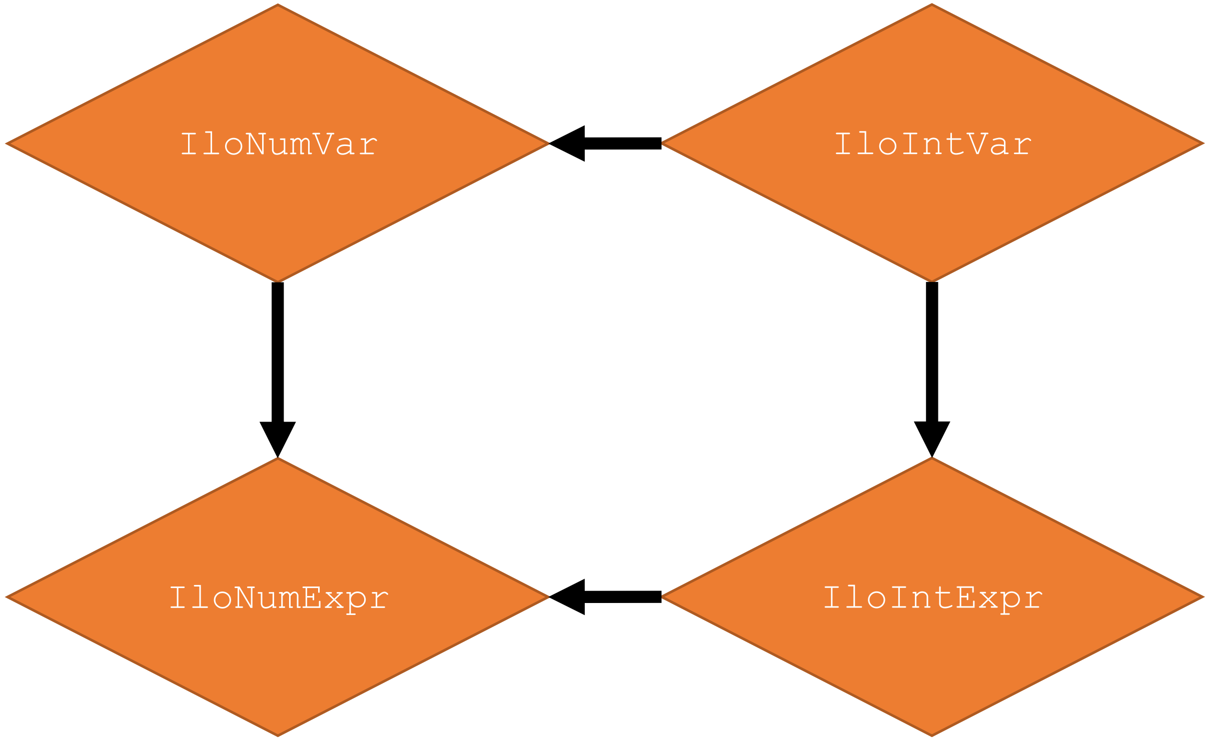
The bounds of a variable can be changed after the variable has been constructed, but its type remains fixed throughout the lifetime of the variable object.

See Also:

`IloNumVarType`, `IloIntVar`



CPLEX Class Hierarchy



CPLEX Class Hierarchy

Interface IloAddable

All Known Subinterfaces:

IloAnd, IloConstraint, IloConversion, IloIntVar, IloLPMatrix, IloModel, IloModeler, IloMPModeler, IloNumVar, IloNumVarBound, IloObjective, IloOr, IloRange, IloSemiContVar, IloSOS1, IloSOS2

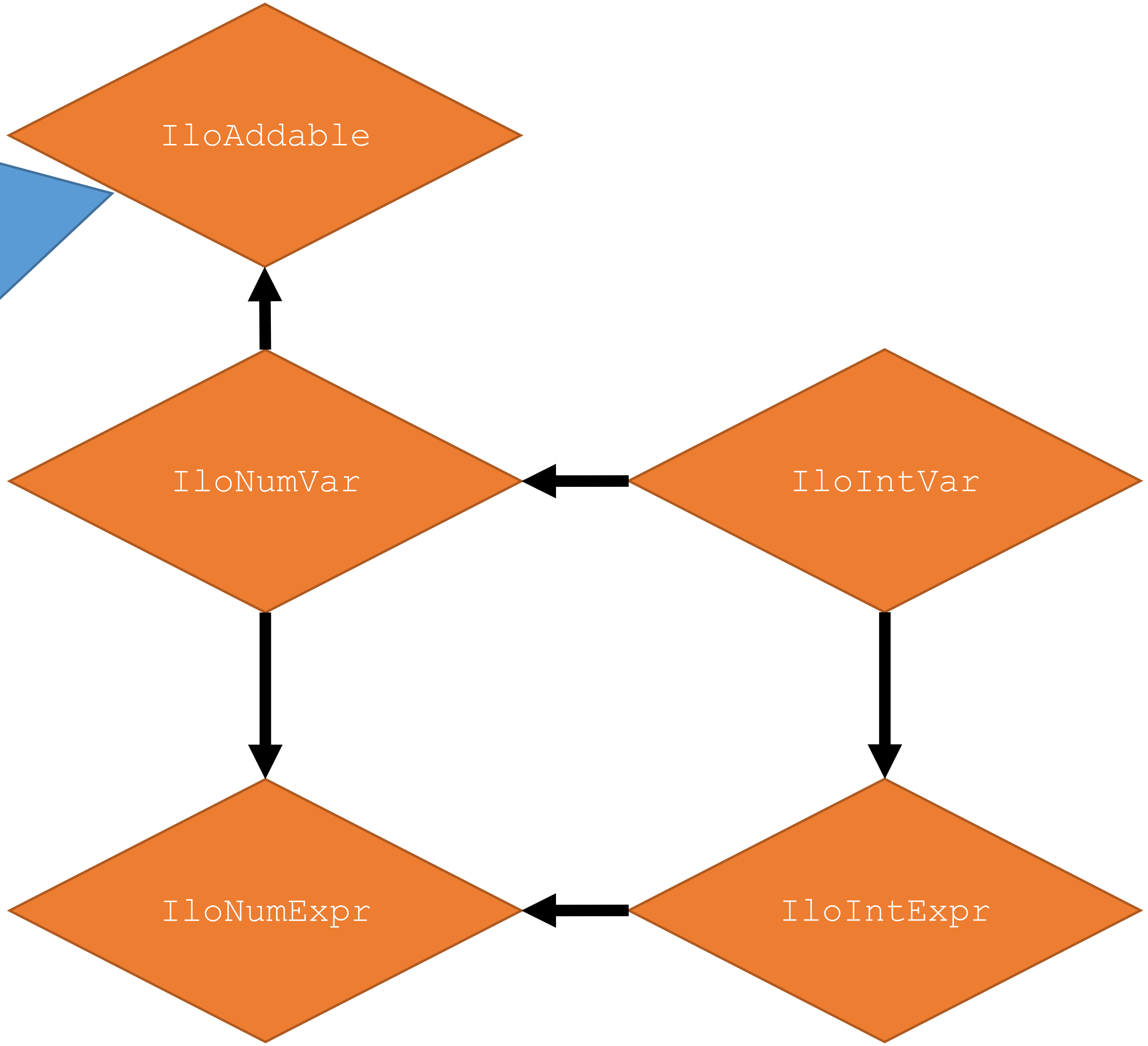
All Known Implementing Classes:

IloCplex, IloCplexModeler

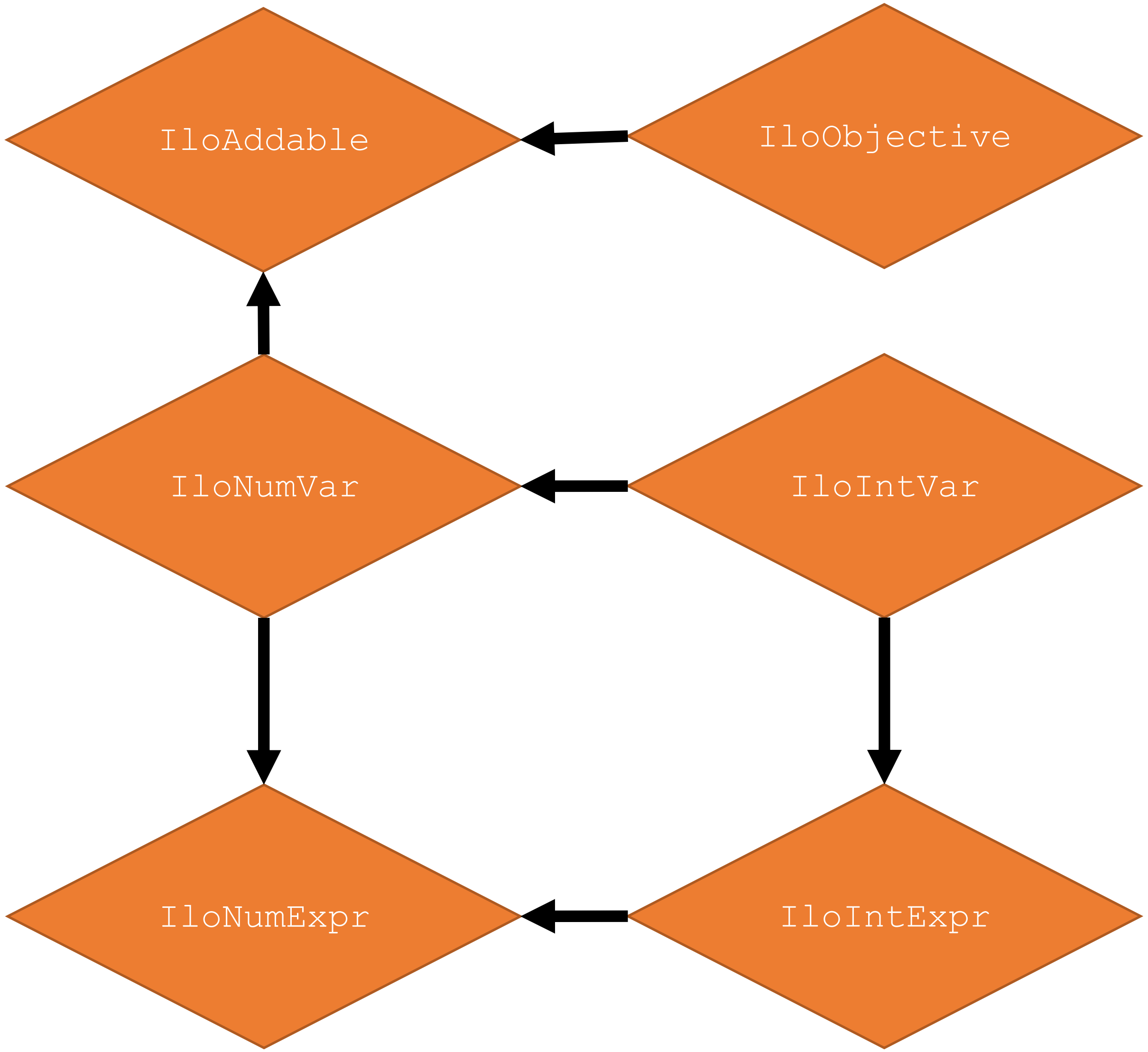
`public interface IloAddable`

This interface is used for modeling objects. Objects of classes implementing this interface can be added to an instance of `IloModel`. Constraint classes, such as `IloRange`, and classes representing optimization objectives, such as `IloObjective`, are examples.

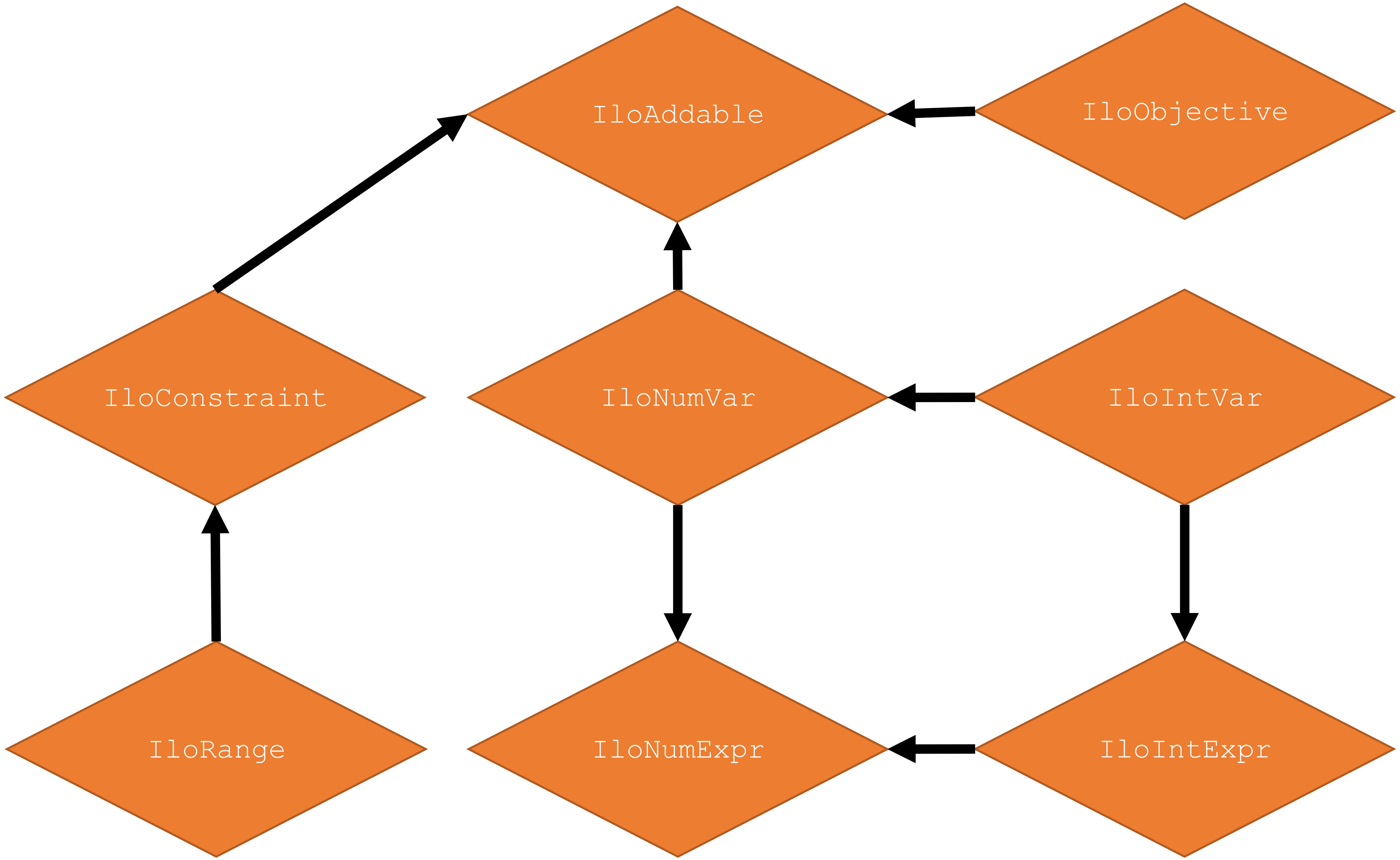
All addable modeling objects can be assigned a name with the method `setName()`. The name can be queried with the method `getName()`. Assigning a name is not required. Modeling objects are created without an assigned name unless a name is specified in the construction.



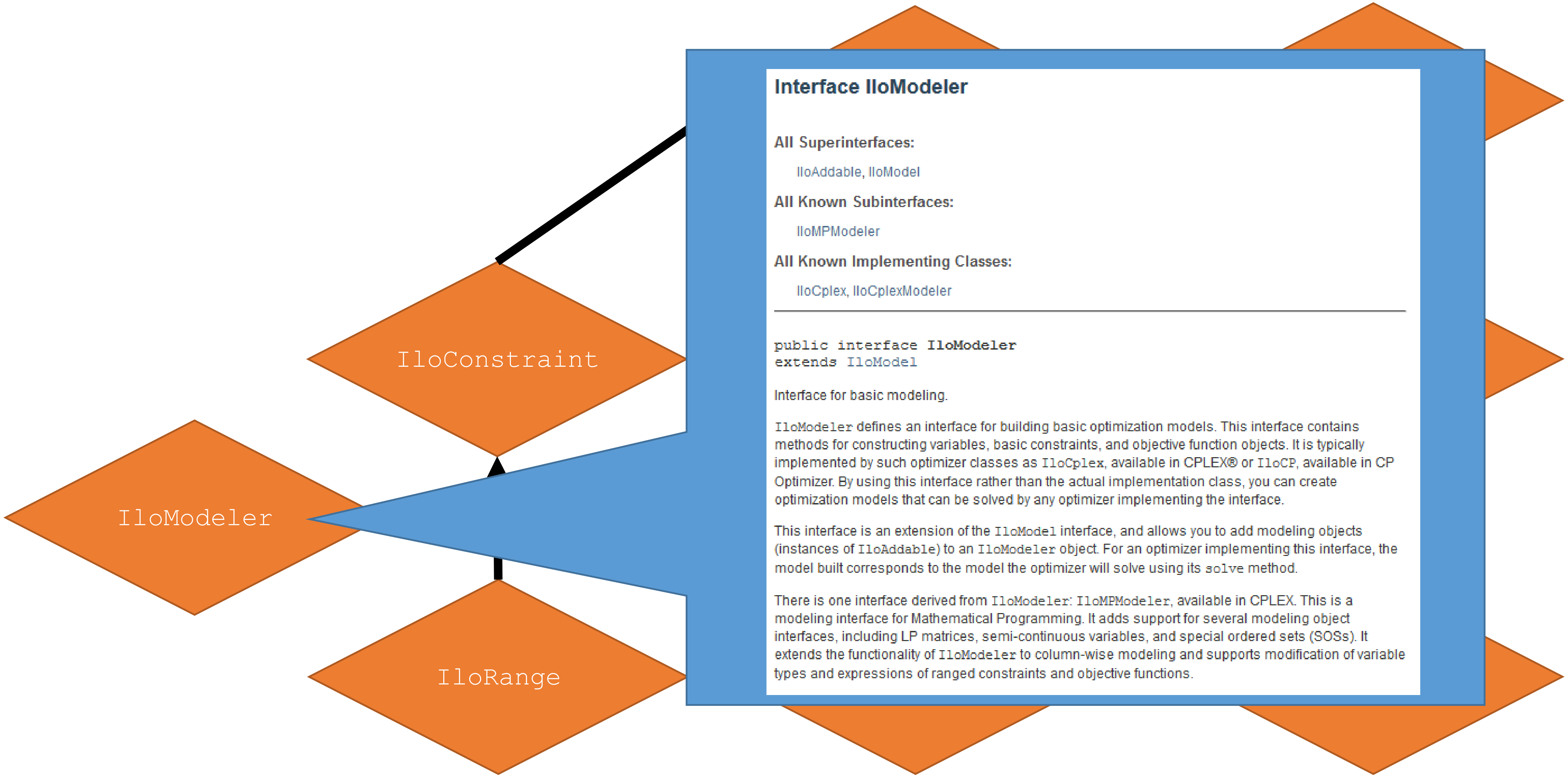
CPLEX Class Hierarchy



CPLEX Class Hierarchy



CPLEX Class Hierarchy



Interface IloModeler

All Superinterfaces:

IloAddable, IloModel

All Known Subinterfaces:

IloMPModeler

All Known Implementing Classes:

IloCplex, IloCplexModeler

```
public interface IloModeler
extends IloModel
```

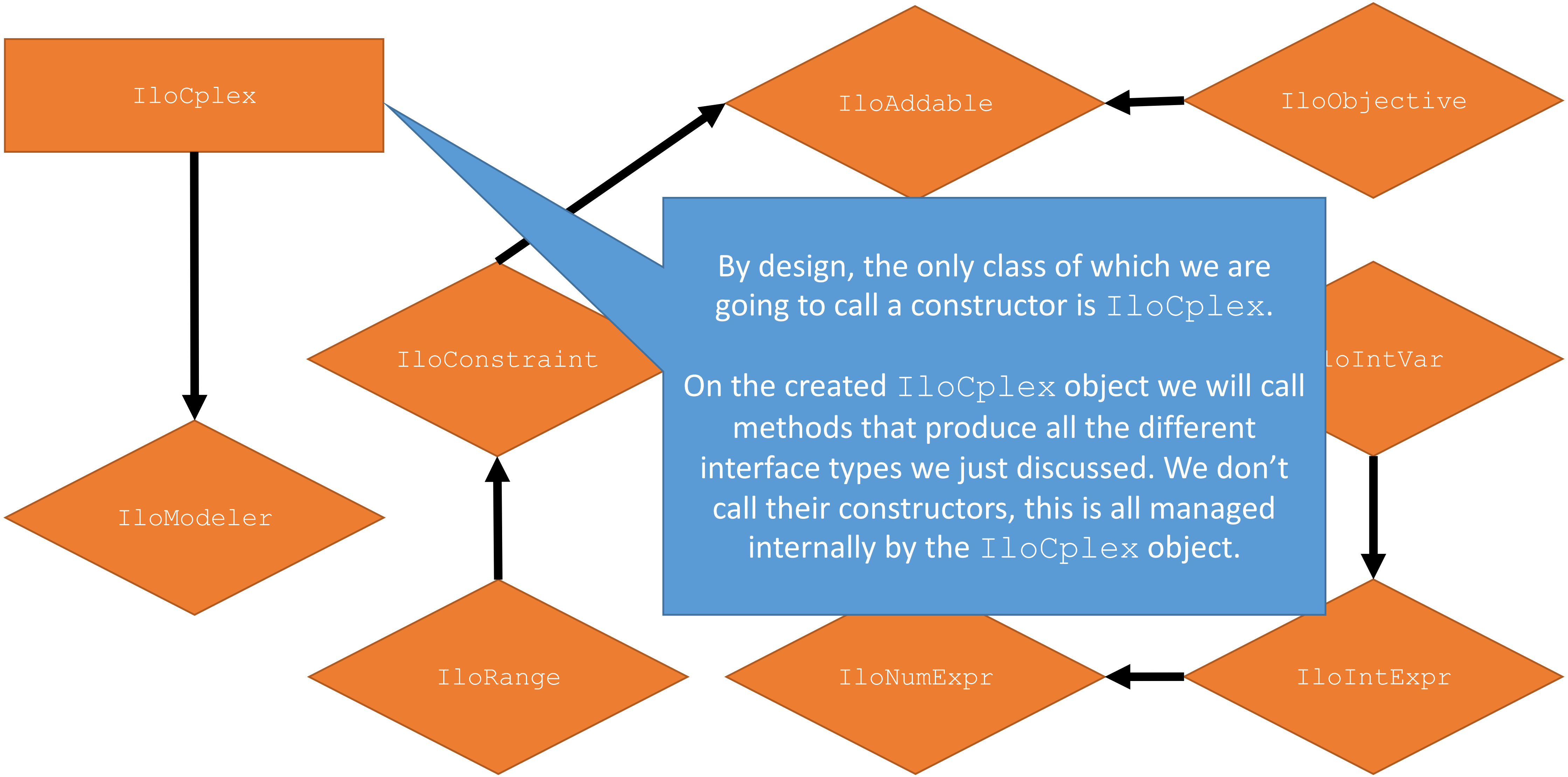
Interface for basic modeling.

`IloModeler` defines an interface for building basic optimization models. This interface contains methods for constructing variables, basic constraints, and objective function objects. It is typically implemented by such optimizer classes as `IloCplex`, available in CPLEX® or `IloCP`, available in CP Optimizer. By using this interface rather than the actual implementation class, you can create optimization models that can be solved by any optimizer implementing the interface.

This interface is an extension of the `IloModel` interface, and allows you to add modeling objects (instances of `IloAddable`) to an `IloModeler` object. For an optimizer implementing this interface, the model built corresponds to the model the optimizer will solve using its `solve` method.

There is one interface derived from `IloModeler`: `IloMPModeler`, available in CPLEX. This is a modeling interface for Mathematical Programming. It adds support for several modeling object interfaces, including LP matrices, semi-continuous variables, and special ordered sets (SOSs). It extends the functionality of `IloModeler` to column-wise modeling and supports modification of variable types and expressions of ranged constraints and objective functions.

CPLEX Class Hierarchy



By design, the only class of which we are going to call a constructor is `IloCplex`.

On the created `IloCplex` object we will call methods that produce all the different interface types we just discussed. We don't call their constructors, this is all managed internally by the `IloCplex` object.

IloCplex – important methods

Variable Creation:

```
IloIntVar   intVar(int min, int max)
IloIntVar   boolVar()
IloNumVar   numVar(double lb, double ub)
```

Building Expressions:

```
IloNumExpr  constant(double c)
IloIntExpr  constant(int c)

IloNumExpr  diff(double v, IloNumExpr e1)
IloIntExpr  diff(IloIntExpr expr1, IloIntExpr expr2)
IloIntExpr  diff(IloIntExpr e, int v)
IloNumExpr  diff(IloNumExpr e, double v)
IloNumExpr  diff(IloNumExpr e1, IloNumExpr e2)
...

IloNumExpr  prod(double v, IloNumExpr e1)
IloIntExpr  prod(IloIntExpr e1, IloIntExpr e2)
IloIntExpr  prod(IloIntExpr e, int v)
IloNumExpr  prod(IloNumExpr e, double v)
...

IloNumExpr  sum(double v, IloNumExpr e)
IloIntExpr  sum(IloIntExpr e1, IloIntExpr e2)
IloIntExpr  sum(IloIntExpr e1, IloIntExpr e2, IloIntExpr e3)
IloIntExpr  sum(IloIntExpr e, int v)
IloNumExpr  sum(IloNumExpr e1, IloNumExpr e2)
...
```

Adding Constraints:

```
IloRange          addEq(IloNumExpr expr, double rhs)
IloConstraint     addEq(IloNumExpr e1, IloNumExpr e2)

IloRange          addGe(IloNumExpr expr, double rhs)
IloConstraint     addGe(IloNumExpr e1, IloNumExpr e2)

IloRange          addLe(IloNumExpr expr, double rhs)
IloConstraint     addLe(IloNumExpr e1, IloNumExpr e2)
```

Adding an objective function:

```
IloObjective      addMaximize(IloNumExpr expr)
IloObjective      addMinimize(IloNumExpr expr)
```

Solution Management:

```
double      getObjValue()
double      getValue(IloNumExpr expr)
double      getDual(IloRange rng)
boolean     solve()
void        writeSolution(String name)
void        readSolution(String name)
```

Model Management Options:

```
void        clearModel()
void        importModel(String name)
void        exportModel(String name)
void        setOut(OutputStream s)
```

IloCplex – important methods

Variable Creation:
```
IloIntVar   intVar(int min, int max)
IloIntVar   boolVar()
IloNumVar   numVar(double lb, double ub)
```

Building Expressions:
```
IloNumExpr  constant(double c)
IloIntExpr  constant(int c)

IloNumExpr  diff(double v, IloNumExpr e1)
IloIntExpr  diff(IloIntExpr expr1, IloIntExpr expr2)
IloIntExpr  diff(IloIntExpr e, int v)
IloNumExpr  diff(IloNumExpr e, double v)
IloNumExpr  diff(IloNumExpr e1, IloNumExpr e2)
...

IloNumExpr  prod(double v, IloNumExpr e1)
IloIntExpr  prod(IloIntExpr e1, IloIntExpr e2)
IloIntExpr  prod(IloIntExpr e, int v)
IloNumExpr  prod(IloNumExpr e, double v)
...

IloNumExpr  sum(double v, IloNumExpr e)
IloIntExpr  sum(IloIntExpr e1, IloIntExpr e2)
IloIntExpr  sum(IloIntExpr e1, IloIntExpr e2, IloIntExpr e3)
IloIntExpr  sum(IloIntExpr e, int v)
IloNumExpr  sum(IloNumExpr e1, IloNumExpr e2)
...
```

Adding Constraints:
```
IloRange        addEq(IloNumExpr expr, double rhs)
IloConstraint   addEq(IloNumExpr e1, IloNumExpr e2)

IloRange        addGe(IloNumExpr expr, double rhs)
IloConstraint   addGe(IloNumExpr e1, IloNumExpr e2)

                double rhs)
                NumExpr e2)

                expr)
                expr)
```

Many more variants available. Have a look at the documentation to figure out what you can and can't do.

There are also useful method if you prefer to work with arrays of variables, in a style that closer resembles working with vectors.

```
boolean   solve()
void      writeSolution(String name)
void      readSolution(String name)
```

Model Management Options:
```
void      clearModel()
void      importModel(String name)
void      exportModel(String name)
void      setOut(OutputStream s)
```

Solving an LP: Steps Required

1. Read instance data from a file into an instance data structure
2. Convert the graph data structure to a CPLEX model
3. Solve the CPLEX model and report the results

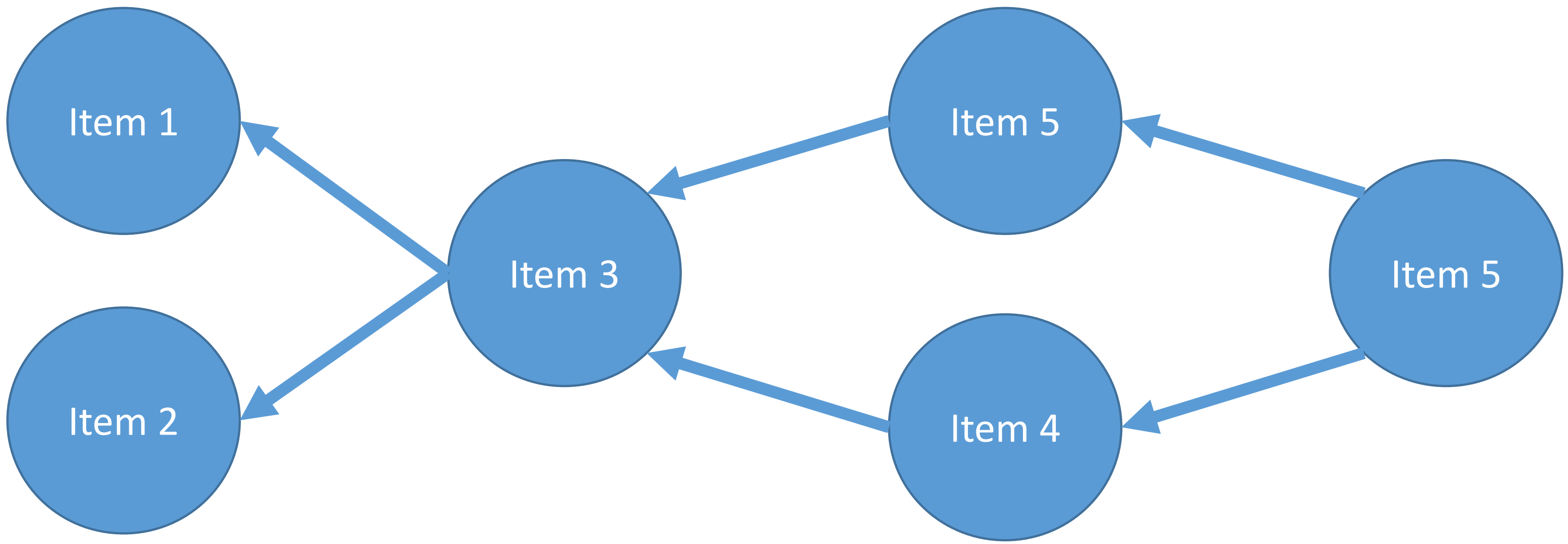
Example:
Precedence Constrained Knapsack Problem

Linear Programming - PCKP

- Often when we describe a linear programming model for a certain problem, we don’t use the matrix notation.
- Example: **The Precedence Constrained Knapsack Problem** (PCKP)
- **Input**:
  - A list of n items, each with a weight w_i and a profit p_i
  - A capacity of the knapsack, b
  - A directed precendence graph G defined on the items
- **Output:**
  - A selection of items such that the sum of their profits is maximized, while the sum of their weights does not exceed the capacity. Items can only be included if **all** its successors in G are also selected.

Linear Programming – PCKP

- Example: **The Precedence Constrained Knapsack Problem**.



- Item 3 can only be included if both Item 1 and 2 are included.
- Item 5 and Item 4 can only be included if Item 3 is included.
- Item 5 can only be included if both Item 4 and Item 5 are included.

Linear Programming – PCKP

- Example: **The Precedence Constrained Knapsack Problem**.
- **Input**:
  - A list of n items, each with a weight w_i and a profit p_i
  - A capacity of the knapsack, b
  - A directed precendence graph G defined on the items
- **Output:**
  - A selection of items such that the sum of their profits is maximized, while the sum of their weights does not exceed the capacity. Items can only be included if **all** its successors in G are also selected.

Maximize
$$\sum_{i=1}^{n} p_i x_i$$
subject to:
$$\sum_{i=1}^{n} w_i x_i \leq b$$
$$x_i \leq x_j \qquad \forall (i,j) \in G$$
$$x_i \in \{0,1\} \qquad \forall i \in \{1, \ldots, n\}$$

Solving an LP: Steps Required

1. **Read instance data from a file into an instance data structure**
2. Convert the graph data structure to a CPLEX model
3. Solve the CPLEX model and report the results

The Item class

```java
public class Item
{
   private final int profit;
   private final int weight;

   public Item(int profit, int weight)
   {
      super();
      this.profit = profit;
      this.weight = weight;
   }

   public int getProfit()
   {
      return profit;
   }

   public int getWeight()
   {
      return weight;
   }

   @Override
   public String toString()
   {
      return "Item [profit=" + profit
            + ", weight=" + weight + "]";
   }
}
```

The `DirectedGraph` class

```
public class DirectedGraph<V,A>
{
   …

   public DirectedGraph() {…}

   public void addNode(V node) throws IllegalArgumentException {…}
   public void addArc(V from, V to, A arcData) throws IllegalArgumentException {…}
   public List<V> getNodes() {…}
   public List<DirectedGraphArc<V,A>> getArcs() {…}
   public List<DirectedGraphArc<V,A>> getOutArcs(V node) throws IllegalArgumentException {…}
   public List<DirectedGraphArc<V,A>> getInArcs(V node) throws IllegalArgumentException {…}
   public int getNumberOfNodes() {…}
   public int getNumberOfArcs() {…}
   public int getInDegree(V node) throws IllegalArgumentException {…}
   public int getOutDegree(V node) throws IllegalArgumentException {…}
}
```

- This is a general class for directed graphs. We can associate different kinds of data with the nodes and the arcs.
    - For example `Integer`’s or `String`’s: `DirectedGraph<Integer,String>`
    - But also: `DirectedGraph<MyNode,MyArc>` (probably necessary for the assignment)

The DirectedGraphArc class

```
public class DirectedGraphArc<V,A>
{
   private final V from;
   private final V to;
   private final A data;

   public DirectedGraphArc(V from, V to, A data)
   {
      this.from = from;
      this.to = to;
      this.data = data;
   }

   public V getFrom()
   {
      return from;
   }

   public V getTo()
   {
      return to;
   }

   public A getData()
   {
      return data;
   }
}
```

Reading instance data

```
3
10 5
7 2
15 7

2
2 0 expensive
0 1 cheap
```

At the beginning of the file, put a number that states how many items will follow. In this example, we have 3 items.

Reading instance data

```
3
10 5
7 2
15 7

2
2 0 expensive
0 1 cheap
```

Each item consists of two data elements (in this example). This line contains the first item.

Reading instance data

```
3
10 5
7 2
15 7

2
2 0 expensive
0 1 cheap
```

After we defined all the items, we also define the relations between the items.
In this example we have two relationships.

Reading instance data

```
3
10 5
7 2
15 7

2
2 0 expensive
0 1 cheap
```

We use three data pieces to model a relationship: two integers and a description.

Reading instance data

```
3
10 5
7 2
15 7

2
2 0 expensive
0 1 cheap
```

```
public static DirectedGraph<Item,String> read(File f) throws FileNotFoundException
{
   try (Scanner scan = new Scanner(f))
   {
      DirectedGraph<Item,String> result = new DirectedGraph<>();
      List<Item> items = new ArrayList<>();

      // Reading the items
      int numItems = scan.nextInt();
      for (int i=0; i < numItems; i++)
      {
          int profit = scan.nextInt();
          int weight = scan.nextInt();
          Item item = new Item(profit,weight);
          items.add(item);
          result.addNode(item);
      }

      // Reading the arcs / precedence constraints
      int numArcs = scan.nextInt();
      for (int i=0; i < numArcs; i++)
      {
         int fromIndex = scan.nextInt();
         int toIndex = scan.nextInt();
         String reason = scan.next();

         Item from = items.get(fromIndex);
         Item to = items.get(toIndex);
         result.addArc(from, to, reason);
      }

      return result;
   }
}
```

Reading instance data

```
3
10 5
7 2
15 7

2
2 0 expensive
0 1 cheap
```

```java
public static DirectedGraph<Item,String> read(File f) throws FileNotFoundException
{
    try (Scanner scan = new Scanner(f))
    {
        DirectedGraph<Item,String> result = new DirectedGraph<>();
        List<Item> items = new ArrayList<>();

        // Reading the items
        int numItems = scan.nextInt();
        for (int i=0; i < numItems; i++)
        {
            int profit = scan.nextInt();
            int weight = scan.nextInt();
            Item item = new Item(profit,weight);
            items.add(item);
            result.addNode(item);
        }

        // Reading the arcs / precedence constraints
        int numArcs = scan.nextInt();
        for (int i=0; i < numArcs; i++)
        {
            int fromIndex = scan.nextInt();
            int toIndex = scan.nextInt();
            String reason = scan.next();

            Item from = items.get(fromIndex);
            Item to = items.get(toIndex);
            result.addArc(from, to, reason);
        }

        return result;
    }
}
```

Reading instance data

```
3
10 5
7 2
15 7

2
2 0 expensive
0 1 cheap
```

```java
public static DirectedGraph<Item,String> read(File f) throws FileNotFoundException
{
   try (Scanner scan = new Scanner(f))
   {
      DirectedGraph<Item,String> result = new DirectedGraph<>();
      List<Item> items = new ArrayList<>();

      // Reading the items
      int numItems = scan.nextInt();
      for (int i=0; i < numItems; i++)
      {
          int profit = scan.nextInt();
          int weight = scan.nextInt();
          Item item = new Item(profit,weight);
          items.add(item);
          result.addNode(item);
      }

      // Reading the arcs / precedence constraints
      int numArcs = scan.nextInt();
      for (int i=0; i < numArcs; i++)
      {
         int fromIndex = scan.nextInt();
         int toIndex = scan.nextInt();
         String reason = scan.next();

         Item from = items.get(fromIndex);
         Item to = items.get(toIndex);
         result.addArc(from, to, reason);
      }

      return result;
   }
}
```

Reading instance data

```
3
10 5
7 2
15 7

2
2 0 expensive
0 1 cheap
```

```java
public static DirectedGraph<Item,String> read(File f) throws FileNotFoundException
{
   try (Scanner scan = new Scanner(f))
   {
      DirectedGraph<Item,String> result = new DirectedGraph<>();
      List<Item> items = new ArrayList<>();

      // Reading the items
      int numItems = scan.nextInt();
      for (int i=0; i < numItems; i++)
      {
         int profit = scan.nextInt();
         int weight = scan.nextInt();
         Item item = new Item(profit,weight);
         items.add(item);
         result.addNode(item);
      }

      // Reading the arcs / precedence constraints
      int numArcs = scan.nextInt();
      for (int i=0; i < numArcs; i++)
      {
        int fromIndex = scan.nextInt();
        int toIndex = scan.nextInt();
        String reason = scan.next();

        Item from = items.get(fromIndex);
        Item to = items.get(toIndex);
        result.addArc(from, to, reason);
      }

      return result;
   }
}
```

Reading instance data

```
3
10 5
7 2
15 7

2
2 0 expensive
0 1 cheap
```

```java
public static DirectedGraph<Item,String> read(File f) throws FileNotFoundException
{
   try (Scanner scan = new Scanner(f))
   {
      DirectedGraph<Item,String> result = new DirectedGraph<>();
      List<Item> items = new ArrayList<>();

      // Reading the items
      int numItems = scan.nextInt();
      for (int i=0; i < numItems; i++)
      {
          int profit = scan.nextInt();
          int weight = scan.nextInt();
          Item item = new Item(profit,weight);
          items.add(item);
          result.addNode(item);
      }

      // Reading the arcs / precedence constraints
      int numArcs = scan.nextInt();
      for (int i=0; i < numArcs; i++)
      {
         int fromIndex = scan.nextInt();
         int toIndex = scan.nextInt();
         String reason = scan.next();

         Item from = items.get(fromIndex);
         Item to = items.get(toIndex);
         result.addArc(from, to, reason);
      }

      return result;
   }
}
```

Model Class

Solving an LP: Steps Required

1. Read instance data from a file into an instance data structure
2. **Convert the graph data structure to a CPLEX model**
3. Solve the CPLEX model and report the results

The `Model` class – Constructor and Variables

```
public class Model
{
   private DirectedGraph<Item,String> instance;
   private int capacity;




   public Model(DirectedGraph<Item,String> instance, int capacity) throws IloException
   {
      // Initialize the instance variables
      this.instance = instance;
      this.capacity = capacity;




   }
   …
}
```

We start by storing the data of a problem instance. In this case we use two parameters for the instance (in the assignment there will be a separate class)

The `Model` class – Constructor and Variables

```java
public class Model
{
   private DirectedGraph<Item,String> instance;
   private int capacity;

   private IloCplex cplex;



   public Model(DirectedGraph<Item,String> tance, int capacity) throws IloException
   {
      // Initialize the instance variables
      this.instance = instance;
      this.capacity = capacity;
      this.cplex = new IloCplex();







   }
   …
}
```

We also initialize an `IloCplex` object which we will use to construct the model.

Almost everything we do with `IloCplex` can throw an `IloException`, which is a checked exception. In this case we throw it to the caller.

The `Model` class – Constructor and Variables

```
public class Model
{
   private DirectedGraph<Item,String> instance;
   private int capacity;

   private IloCplex cplex;



   public Model(DirectedGraph<Item,String> instance, int capacity) throws IloException
   {
      // Initialize the instance variables
      this.instance = instance;
      this.capacity = capacity;
      this.cplex = new IloCplex();








   }
   …
}
```

We also initialize an `IloCplex` object which we will use to construct the model.

Almost everything we do with `IloCplex` can throw an `IloException`, which is a checked exception. In this case we throw it to the caller.

The `Model` class – Constructor and Variables

```java
public class Model
{
   private DirectedGraph<Item,String> instance;
   private int capacity;

   private IloCplex cplex;

   private Map<Item,IloNumVar> varMap;

   public Model(DirectedGraph<Item,String> instance, int capacity) throws IloException
   {
      // Initialize the instance variables
      this.instance = instance;
      this.capacity = capacity;
      this.cplex = new IloCplex();

      // Create a map to link items to variables
      this.varMap = new HashMap<>();




   }
   …
}
```

It is often very useful to be able to translate between variables in the model and objects from our problem instance. For this purpose we create a `Map` that can translate an Item object to a decision variable.

The `Model` class – Constructor and Variables

```
public class Model
{
   private DirectedGraph<Item,String> instance;
   private int capacity;

   private IloCplex cplex;

   private Map<Item,IloNumVar> varMap;

   public Model(DirectedGraph<Item,String> instance, int capacity) throws IloException
   {
      // Initialize the instance variables
      this.instance = instance;
      this.capacity = capacity;
      this.cplex = new IloCplex();

      // Create a map to link items to variables
      this.varMap = new HashMap<>();




   }
   …
}
```

It is often very useful to be able to translate between variables in the model and objects from our problem instance. For this purpose we create a `Map` that can translate an Item object to a decision variable.

The `Model` class – Constructor and Variables

```java
public class Model
{
   private DirectedGraph<Item,String> instance;
   private int capacity;

   private IloCplex cplex;

   private Map<Item,IloNumVar> varMap;

   public Model(DirectedGraph<Item,String> instance, int capacity) throw
   {
      // Initialize the instance variables
      this.instance = instance;
      this.capacity = capacity;
      this.cplex = new IloCplex();

      // Create a map to link items to variables
      this.varMap = new HashMap<>();

      // Initialize the model. It is important to initialize the variables first!
      addVariables();
      addKnapsackConstraint();
      addPrecedenceConstraints();
      addObjective();
   }
   …
}
```

In general it is a good idea to do the model building in separate methods. Start with a method for initializing the variables.

Add a separate method for each type of constraint. This way it is easy to disable one type of constraint by commenting out the call to the initialization function.

Finally, call a method that will initialize the objective.

The `Model` class – Model Building

```
public class Model
{
……
   private void addVariables() throws IloException
   {
      for (Item i : instance.getNodes())
      {
         IloNumVar var = cplex.boolVar();
         varMap.put(i, var);
      }
   }
……
}
```

In this case we create binary variables of and put them into the map.

We do nothing else with the variables, as we will only use the in the other initialization methods.

If we want to create a model that allows for fractionally selected items, we should call `cplex.numVar(0,1);`

$$x_i \in \{0,1\}$$

The `Model` class – Model Building

```java
public class Model
{
……
   private void addKnapsackConstraint() throws IloException
   {
      IloNumExpr lhs = cplex.constant(0);






   }
……
}
```

As we want to create an expression that is the sum of every binary variable multiplied by the weight of the corresponding item, it is a good idea to start an "empty" expression with the constant 0, as adding 0 to any expression does not change it.

$$\sum_{i=1}^{n} a_i x_i \leq b$$

The `Model` class – Model Building

```
public class Model
{
……
   private void addKnapsackConstraint() throws IloException
   {
      IloNumExpr lhs = cplex.constant(0);
      for (Item i : instance.getNodes())
      {
         IloNumVar var = varMap.get(i);
         IloNumExpr term = cplex.prod(i.getWeight(), var);

      }

   }
……
}
```

For every item, we retrieve the corresponding variable and create a term by taking its product with the weight of the item.

$$\sum_{i=1}^{n} a_i x_i \leq b$$

The `Model` class – Model Building

```java
public class Model
{
……
   private void addKnapsackConstraint() throws IloException
   {
      IloNumExpr lhs = cplex.constant(0);
      for (Item i : instance.getNodes())
      {
         IloNumVar var = varMap.get(i);
         IloNumExpr term = cplex.prod(i.getWeight(), var);
         lhs = cplex.sum(lhs, term);
      }

   }
……
}
```

We then add the term to the expression we had build until now.

$$\sum_{i=1}^{n} a_i x_i \leq b$$

The `Model` class – Model Building

```
public class Model
{
……
   private void addKnapsackConstraint() throws IloException
   {
      IloNumExpr lhs = cplex.constant(0);
      for (Item i : instance.getNodes())
      {
         IloNumVar var = varMap.get(i);
         IloNumExpr term = cplex.prod(i.getWeight(), var);
         lhs = cplex.sum(lhs, term);
      }
      cplex.addLe(lhs, capacity);
   }
……
}
```

Finally, we add the expression as a constraint to the model, by stating that the expression should be smaller than the capacity.

$$\sum_{i=1}^{n} a_i x_i \leq b$$

The `Model` class – Model Building

```
public class Model
{
……
   private void addPrecedenceConstraints() throws IloException
   {
      for (DirectedGraphArc<Item,String> arc : instance.getArcs())
      {
         IloNumVar from = varMap.get(arc.getFrom());
         IloNumVar to = varMap.get(arc.getTo());
         cplex.addLe(from, to);
      }
   }
……
}
```

For the precedence constraints, we iterate over the arcs in the graph and create a new constraint for each arc.

$$x_i \leq x_j$$

The `Model` class – Model Building

```java
public class Model
{
……
   private void addPrecedenceConstraints() throws IloException
   {
      for (DirectedGraphArc<Item,String> arc : instance.getArcs())
      {
         IloNumVar from = varMap.get(arc.getFrom());
         IloNumVar to = varMap.get(arc.getTo());
         cplex.addLe(from, to);
      }
   }
……
}
```

For the precedence constraints, we iterate over the arcs in the graph and create a new constraint for each arc.

$$x_i \leq x_j$$

The `Model` class – Model Building

```java
public class Model
{
……
   private void addObjective() throws IloException
   {
      IloNumExpr obj = cplex.constant(0);
      for (Item i : instance.getNodes())
      {
         IloNumVar var = varMap.get(i);
         IloNumExpr term = cplex.prod(var, i.getProfit());
         obj = cplex.sum(obj, term);
      }

   }
……
}
```

Building the expression for the objective happens in a very similar way to building the expression for the constraint, except we now use `getProfit()` instead of `getWeight()`

Maximize $\sum_{i=1}^{n} p_i x_i$

The `Model` class – Model Building

```java
public class Model
{
……
   private void addObjective() throws IloException
   {
      IloNumExpr obj = cplex.constant(0);
      for (Item i : instance.getNodes())
      {
         IloNumVar var = varMap.get(i);
         IloNumExpr term = cplex.prod(var, i.getProfit());
         obj = cplex.sum(obj, term);
      }
      cplex.addMaximize(obj);
   }
……
}
```

We call `addMaximize()` to add the objective function.
If we would like to modify the objective later, we should store the `IloObjective` object, but in this case we don't need it for later.

Maximize $$\sum_{i=1}^{n} p_i x_i$$

Solving and Updating the Model

Solving an LP: Steps Required

1. Read instance data from a file into an instance data structure
2. Convert the graph data structure to a CPLEX model
3. **Solve the CPLEX model and report the results**

The `Model` class – Solving and the Solution

```
public class Model
{
……
   public boolean solve() throws IloException
   {
      return cplex.solve();
   }









……
}
```

For solving the current model, we can call the `solve()` method on the `IloCplex` object.

This method returns true if a solution was found and false if not (for example because it is infeasible)

The `Model` class – Solving and the Solution

```
public class Model
{
……
    public boolean solve() throws IloException
    {
        return cplex.solve();
    }

    public List<Item> getSolution() throws IloException
    {
        List<Item> result = new ArrayList<>();







        return result;
    }
……
}
```

We create a method that will take the values of the decision variables and builds a list of selected items.

The `Model` class – Solving and the Solution

```
public class Model
{
……
   public boolean solve() throws IloException
   {
      return cplex.solve();
   }

   public List<Item> getSolution() throws IloException
   {
      List<Item> result = new ArrayList<>();
      for (Item i : instance.getNodes())
      {
         IloNumVar var = varMap.get(i);
         double value = cplex.getValue(var);




      }
      return result;
   }
……
}
```

We can also obtain the value of a variable in the solution using the `getValue()` method and passing relevant `IloNumVar` objects as an argument.

The `Model` class – Solving and the Solution

```
public class Model
{
……
   public boolean solve() throws IloException
   {
      return cplex.solve();
   }

   public List<Item> getSolution() throws IloException
   {
      List<Item> result = new ArrayList<>();
      for (Item i : instance.getNodes())
      {
         IloNumVar var = varMap.get(i);
         double value = cplex.getValue(var);
         if (value >= 0.5)
         {
            result.add(i);
         }
      }
      return result;
   }
……
}
```

We have to be careful with numerical precision: decision variables may get a value that is slightly less than 1. Since we work with integer solutions, we can use 0.5 as a threshold. For continuous variables, we have to use threshold closer to 1.

Solving the Model

```java
public static void main(String [] args)
{
   try
   {
      DirectedGraph<Item,String> instance = read(new File("instance.txt"));
      System.out.println("The following instance was read:");
      System.out.println(instance);

      Model model = new Model(instance, 9);
      model.solve();
      System.out.println(model.getSolution());
   }
   catch (IloException e)
   {
      e.printStackTrace();
   }
   catch (FileNotFoundException e)
   {
      e.printStackTrace();
   }
}
```

```
The following instance was read:
DirectedGraph [nodes=[Item [profit=10, weight=5], Item [profit=7, weight=2], Item [profit=15,
weight=7]], arcs=[Arc [from=Item [profit=15, weight=7], to=Item [profit=10, weight=5],
data=expensive], Arc [from=Item [profit=10, weight=5], to=Item [profit=7, weight=2],
data=cheap]]]

Found incumbent of value 0.000000 after 0.00 sec. (0.00 ticks)
Tried aggregator 1 time.
MIP Presolve eliminated 3 rows and 3 columns.
MIP Presolve modified 3 coefficients.
All rows and columns eliminated.
Presolve time = 0.00 sec. (0.00 ticks)

Root node processing (before b&c):
  Real time             =    0.00 sec. (0.01 ticks)
Parallel b&c, 2 threads:
  Real time             =    0.00 sec. (0.00 ticks)
  Sync time (average)   =    0.00 sec.
  Wait time (average)   =    0.00 sec.
                          ------------
Total (root+branch&cut) =    0.00 sec. (0.01 ticks)

[Item [profit=10, weight=5], Item [profit=7, weight=2]]
```

Updating the Model

- After solving the model, we may want to update it and solve again.

```
public class Model
{
……
public void setItem(Item i, boolean enabled) throws IloException
{
      IloNumVar var = varMap.get(i);
      if (enabled)
      {
         var.setLB(0);
         var.setUB(1);
      }
      else
      {
         var.setLB(0);
         var.setUB(0);
      }
   }
……
}
```

If we set the lower bound and upper bound of a variable to the same value, it is fixed to that value.

Updating the Model

```java
public static void main(String [] args)
{
   try
   {
      DirectedGraph<Item,String> instance = read(new File("instance.txt"));
      System.out.println("The following instance was read:");
      System.out.println(instance);

      Model model = new Model(instance, 9);
      model.solve();
      System.out.println(model.getSolution());

      Item i = instance.getNodes().get(0);
      model.setItem(i, false);
      model.solve();
      System.out.println(model.getSolution());
   }
   catch (IloException e)
   {
      e.printStackTrace();
   }
   catch (FileNotFoundException e)
   {
      e.printStackTrace();
   }
}
```

```
The following instance was read:
DirectedGraph [nodes=[Item [profit=10, weight=5], Item [profit=7, weight=2], Item [profit=15,
weight=7]], arcs=[Arc [from=Item [profit=15, weight=7], to=Item [profit=10, weight=5],
data=expensive], Arc [from=Item [profit=10, weight=5], to=Item [profit=7, weight=2],
data=cheap]]]

[…]

[Item [profit=10, weight=5], Item [profit=7, weight=2]]

[…]

[Item [profit=7, weight=2]]
```

First solution

Updating the Model

```java
public static void main(String [] args)
{
    try
    {
        DirectedGraph<Item,String> instance = read(new File("instance.txt"));
        System.out.println("The following instance was read:");
        System.out.println(instance);

        Model model = new Model(instance, 9);
        model.solve();
        System.out.println(model.getSolution());

        Item i = instance.getNodes().get(0);
        model.setItem(i, false);
        model.solve();
        System.out.println(model.getSolution());
    }
    catch (IloException e)
    {
        e.printStackTrace();
    }
    catch (FileNotFoundException e)
    {
        e.printStackTrace();
    }
}
```

```
The following instance was read:
DirectedGraph [nodes=[Item [profit=10, weight=5], Item [profit=7, weight=2], Item [profit=15, weight=7]], arcs=[Arc [from=Item [profit=15, weight=7], to=Item [profit=10, weight=5], data=expensive], Arc [from=Item [profit=10, weight=5], to=Item [profit=7, weight=2], data=cheap]]]

[…]

[Item [profit=10, weight=5], Item [profit=7, weight=2]]

[…]

[Item [profit=7, weight=2]]
```

Second solution

Final Hints and Tips

Libraries

- To prevent that we have to reinvent the wheel each time, we should use libraries for specific tasks
- The CPLEX library can be used to model (Integer) Linear Programming Problems
- Other very useful libraries exist as well:
  - The Apache Math Commons library has a lot of useful tools for mathematical computations (statistics, probability distributions, etc)
  - Apache POI can be used to read and write Excel files
  - Jackson-databind is useful for reading and writing objects easily
  - countless others
- If you need many libraries, consider learning a dependency manager such as Maven.

Tips For Your Assignment

- Make use of the DirectedGraph and DirectedGraphArc classes provided in the CPLEX example.
- To store demand, supply and the costs, think about which data types you want to assign to the nodes and arcs:
- Most elegant is to write your own data classes and have a DirectedGraph<MyNodeData,MyArcData>
  - If you are lazy, try only declaring instance variables and let Eclipse generate the constructor, getters/setters and toString() method for you.
  - Alternatively you can consider a DirectedGraph<List<Integer>,List<Integer>>
- You can fix variables to 1 or 0 using the lower and upper bounds.
- To debug your CPLEX model, you can export it to a file.