The OptimJ manual

Version 1.3.14

OptimJ is an extension of the Java programming language with language support for writing optimization models and powerful abstractions for bulk data processing. The language is supported by programming tools under the Eclipse environment.

- OptimJ is an extension of Java 5 :
  - OptimJ operates directly on Java objects and can be combined with any other Java classes (as source files or class files).
  - The whole Java library is directly available from OptimJ
  - OptimJ is interoperable with most Java-based programming tools such as unit testing or GUI designers.
- OptimJ is a modeling language :
  - Algebraic modeling concepts found in modeling languages such as AIMMS, AMPL, GAMS, MOSEL, MPL, OPL, etc., are expressible in OptimJ.
  - OptimJ must be backed by an optimization engine offering a Java API: this version supports lp_solve, glpk, Gurobi, Mosek and CPLEX. Other solver interfaces are in preparation, and additional solvers can be used indirectly via the LP and MPS file formats.

This document presents the OptimJ language and programming environment. We expect familiarity of the reader with the Java language, the Eclipse IDE and basic optimization techniques.

► See section 1 for a quick overview of OptimJ features and benefits.
► See section 2 for checking your OptimJ installation and running your first project

The following sections describe the OptimJ language. Then :

► If you're familiar with Java, jump to section 8

OptimJ comes with many samples: the easiest way to write your first OptimJ program is to start with an example and adapt it to your needs.

Legal notices

This document does not imply any kind of contractual commitment from Ateji.

Ateji and OptimJ are trademarks of Ateji SAS. Java is a trademark of Sun Microsystems, Inc. Eclipse is a trademark of Eclipse Foundation, Inc. Mosek is a trademark of Mosek ApS. CPLEX is a trademark of ILOG Inc. Gurobi is a trademark of Gurobi Inc. All other marks belong to their respective owners.

System requirements

- Java Runtime JRE 1.5 or later (http://java.sun.com/)
- Eclipse version 3.6.x (Helios), 3.5.x (Galileo) or 3.4.x (Ganymede) (http://www.eclipse.org)
- One of the supported optimization solvers

Contact

info@ateji.com
www.ateji.com

1 Features and Benefits

OptimJ is a familiar full-featured modeling language:

- It expresses optimization models in a natural mathematical-like notation
- It is solver independent : the language itself can express any optimization model regardless of the underlying solver. Only when committing to a particular solver are the corresponding restrictions (such as linearity) taken into account.
- It permits access to the whole solver API.
- The modeling part of the language features :
    - models, decision variables, constraints
    - aggregate operators
    - generalized declarations
    - global constraints
    - higher-order constraints
    - tuples
- It enables Object-Oriented modeling, with all OO constructs available within the constraints

OptimJ is an extension of the Java programming language :

- Any boolean Java expression involving decision variables is a constraint
- An OptimJ model and a Java application work together with zero integration effort
- The modelization code operates directly over your application data
- All constructs of the Java language are available inside models
- The whole Java library is available inside models
- Additionally, aggregate operators over collections provide an concise and natural way to preprocess data.

OptimJ is integrated in the Eclipse IDE :

- Edit, compile, debug as usual
- Keep your development process unchanged, common programming tools such as Ant, CVS, etc., work with OptimJ sources
- Source-to-source transformation enables the use of Java productivity tools such as metrics and programming guidelines ensurance.

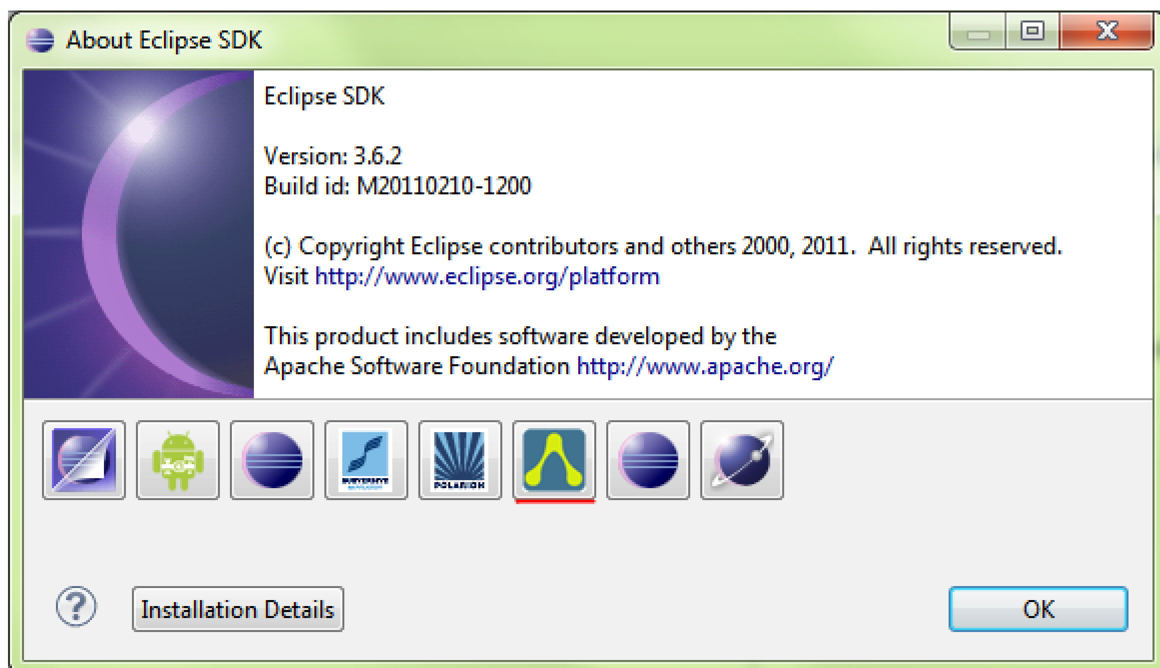
OptimJ works out of the box : you simply install an Eclipse plugin. OptimJ is easy to learn : optimization specialists will find a familiar optimization language, Java specialists only need to learn a few additional constructs, and they communicate using a common language. OptimJ interacts seamlessly with your applications and doesn't require changes in your programming process.

(C) Ateji. All rights reserved.

2 OptimJ Jump Start

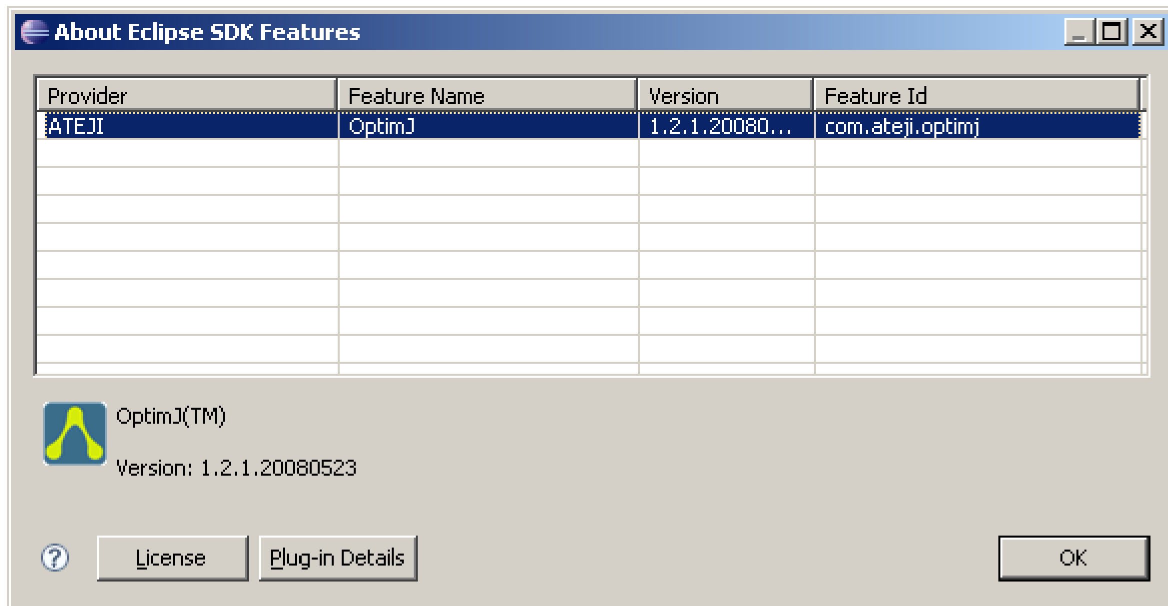
Make sure the OptimJ plugin is correctly installed in your Eclipse IDE

OptimJ requires Eclipse SDK version 3.6 (Helios), 3,5 (Galileo) or 3.4 (Ganymede). Select "Help" -> "About Eclipse SDK" :



Clicking on the Ateji logo gives an alternative way to see the OptimJ plugin details :

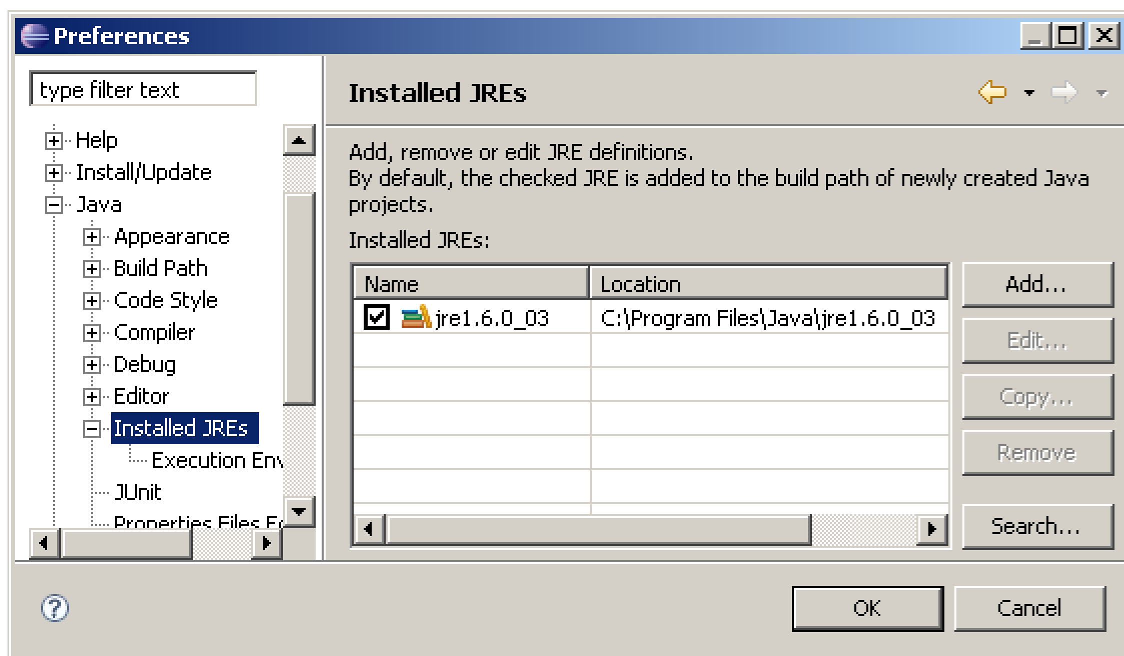
(C) Ateji. All rights reserved.



Make sure your Java compiler matches OptimJ system requirements

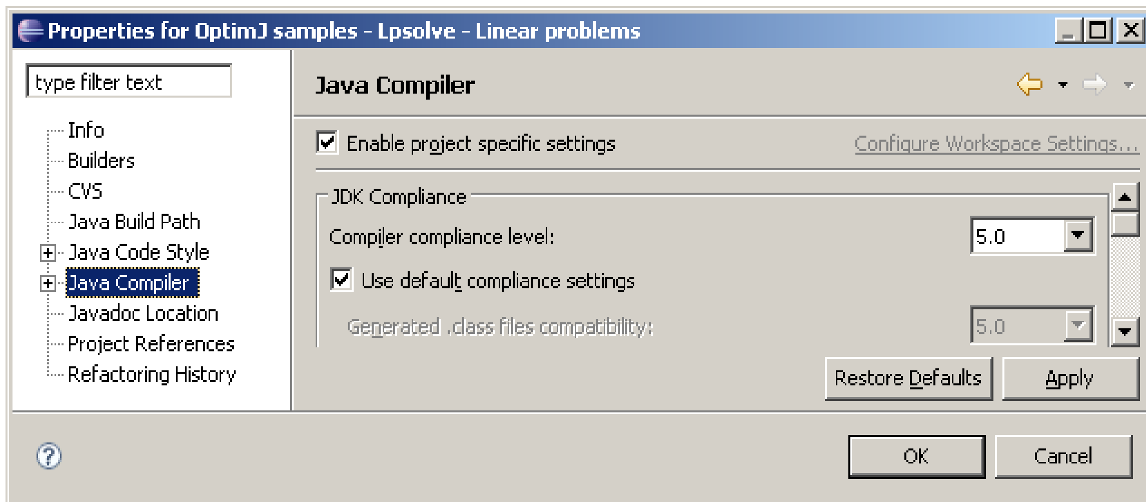
OptimJ requires Java 1.5 or later.

1. Select "Window" -> "Preferences...", then choose "Java" -> "Installed JREs":



2. Then right-click on your project. Select "properties" then choose "java compiler":

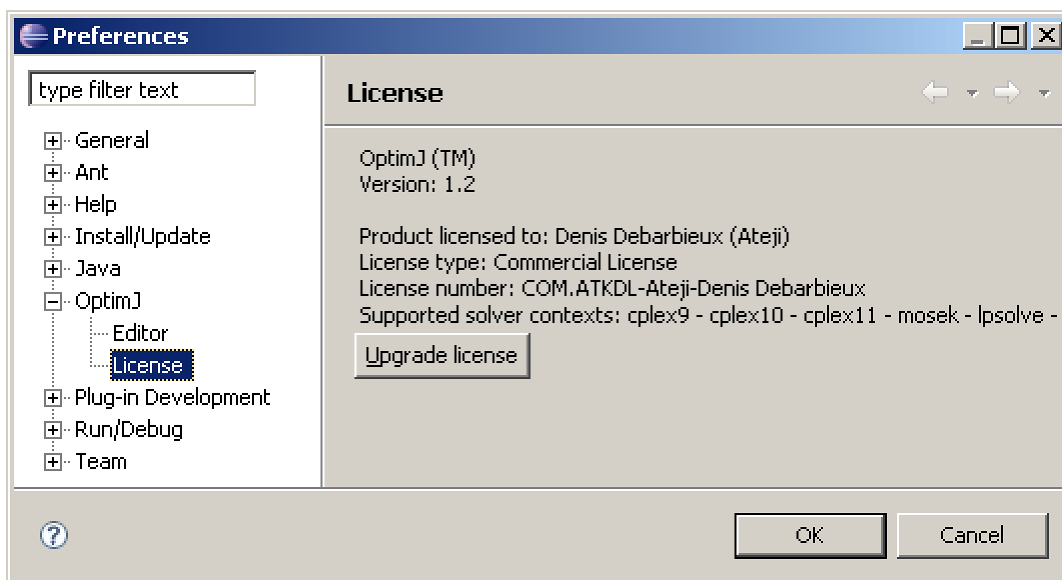
(C) Ateji. All rights reserved.



Make sure you have obtained and entered a licence number :

Select "Window" -> "Preferences...", then choose "OptimJ" -> "License" :

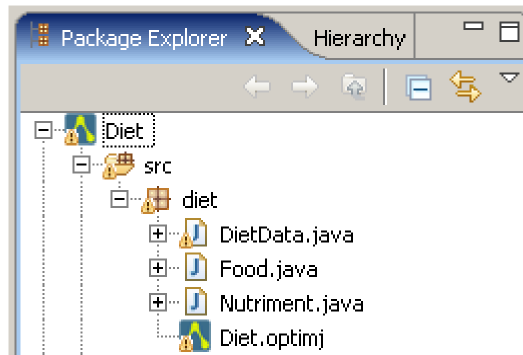
The corresponding entry in the "*Preferences*" window displays your licence informations, including OptimJ version number and the name of the licencee. If you don't have a licence number, contact `sales@ateji.com`.



Open an OptimJ source file in an OptimJ project

(C) Ateji. All rights reserved.

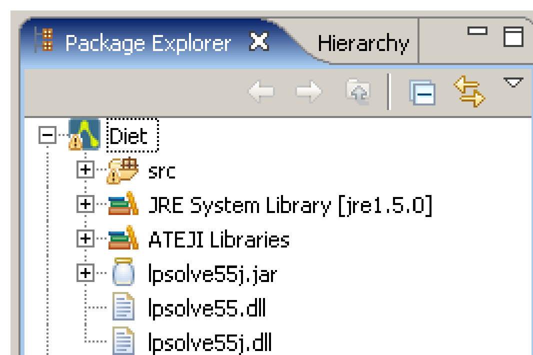
When the OptimJ plugin is properly installed and enabled, OptimJ projects and OptimJ source files (**`.optimj`** files) appear with the Ateji icon :



As your first OptimJ project, we recommend you to download the OptimJ samples available from Ateji web site. See chapter "Installation" for details.

You can also convert one of your existing Java projects into an OptimJ project by adding it the OptimJ nature. Right-click on a Java project and select :

Check that a solver is visible from your build path



The OptimJ sample projects contain distributions of some common freely available solvers.

You're done !

You can now edit, compile, run, debug OptimJ source files as you would with Java source files.

(C) Ateji. All rights reserved.

3 Language constructs for modeling

3.1 The basics : optimization concepts in Java

An OptimJ program combines Java code and optimization code. Here is a basic example:

```
package examples;

// a simple model for the map-coloring problem
public model SimpleColoring solver lpsolve
{
	// maximum number of colors
	int nbColors = 4;

	// decision variables hold the color of each country
	var int belgium in 1 .. nbColors;
	var int denmark in 1 .. nbColors;
	var int germany in 1 .. nbColors;

	// neighbouring countries must have a different color
	constraints {
		belgium != germany;
		germany != denmark;
	}

	// a main entry point to test our model
	public static void main(String[] args)
	{
		// instanciate the model
		SimpleColoring m = new SimpleColoring();

		// solve it
		m.extract();
		m.solve();

		// print solutions
		System.out.println("Belgium: " + m.value(m.belgium));
		System.out.println("Denmark: " + m.value(m.denmark));
		System.out.println("Germany: " + m.value(m.germany));
	}
}
```

First OptimJ program : a simple model for the map coloring problem.

(C) Ateji. All rights reserved.

If you're familiar with Java, you will notice a few additional constructions : **model**, **var**, **constraints**.

If you're familiar with modeling languages, you will notice a few additional programming artefacts such as **package**, **import** and the `main()` method.

Running this example will print one of the possible solutions :

```
Belgium: 1
Denmark: 3
Germany: 2
```

A **model** is a Java class extended with optimization specific concepts. Like any other Java class, it has modifiers for controlling access and visibility, it can extend classes and implement interfaces, it can be referenced in the application everywhere a class can be referenced. Like classes, models have constructors, fields and methods. Models are final classes (they cannot be extended).

Additionally, models define **decision variables** (section 3.4), **constraints** (section 3.5), **objectives** (section 3.7).

3.2 Solver independence and solver integration

OptimJ is a language designed independently of any solver or optimization engine. It can express arbitrarily complex optimization problems, however when tying the language to a solver only models understood by the solver are allowed. As an example, OptimJ allows you to write the following, although not many solvers on the market may be able to solve it directly :

```
abstract model BallisticBody
{
	double g = 9.81;
	var double v0, alpha,range;

	constraints {
		// The ballistic body equation
		range == v0*v0 * Math.sin(2 * alpha) / g ;
	}

	maximize range;
}
```

An abstract model with trigonometric and non-linear constraints

Note the **abstract** keyword : an abstract model is not tied to a particular solver engine, hence there are no restrictions on the models you can write. Obviously, without a solver you won't be able to solve your model. Since the model is abstract, you won't even be able to instanciate it.

Once you provide the name of the solver to be used, your model is not abstract anymore : you'll be able to instanciate and solve it. But the models you can write will be restricted to the models that your solver can handle.

You can think of the compilation of OptimJ as a two-phase process. The first phase is purely on the language level, it ensures that programs are syntactically correct, that the typing rules are verified, etc. The second phase checks that the optimization features used in the program can be handled by the specific solver in use, and rejects those that cannot (most solvers have restrictions on data types and on linearity of constraints). These restrictions are not hard-wired in the language.

(C) Ateji. All rights reserved.

3.3 Solver context

Some concepts such as instanciation of a decision variable or creation of a constraint make sense only in relation with a given solver. In order to handle this restriction, OptimJ introduces the notion of a *solver context*.

A solver context is introduced by the model declaration with the keyword **solver** :

```
model SimpleColoring solver lpsolve
{
    ...
}
```

Introducing a solver context

The scope of the solver context is the fields of the model, the constraints blocks and the objective clause. The solver context cannot be redefined – this is why a model cannot extend a model.

The solver context puts restrictions on the types allowed for decision variables, on the operations allowed for writing constraints, and on the structure of constraints (linear, quadratic, etc.).

The solver context specifies the methods that can be invocated on decision variables. It also defines a "solver object", accessible through the method **solver()**, and the methods that it provides (see section 4). This is OptimJ's way of bringing into the language all the solver specific APIs.

```
// this example shows the parts impacted by the choice of a solver
import lpsolve.LpSolve;

model Coloring solver lpsolve
{
    var double x; // OK, lpsolve handles double variables
    constraints {
        2*x <= 3; // OK, lpsolve handles linear constraints
    }

    public static void main(String[] args)
    {
        // here we instanciate the model : its type is 'Coloring'
        Coloring coloring = new Coloring();
        // extracting the model also instanciates the solver
        coloring.extract();

        // here we get the instance of the solver :
        // its type depends on the choosen solver
        LpSolve mySolver = coloring.solver();
    }
}
```

Introducing a solver context

(C) Ateji. All rights reserved.

The solver context is not user-definable. Each distribution of OptimJ comes accompanied with a separate documentation detailing the capabilities of the solver contexts it supports. The second part of this manual describes the available solver contexts.

Note that the solver context may not only refer to an optimization engine, it can also be the name of a file format (MPS, LP, etc.). Extracting a model in this case means generating the corresponding text file.

3.4 Decision variables

The major difference between an imperative language such as Java and a modeling language is the notion of *what is a variable*.

- Imperative variables are names referring to memory areas, in which one can store and read values.
- Decision variables (sometimes also called logical variables) are placeholders for the solution of a problem. The set of all possible values for the solution must be given in the declaration (either explicitly, or implicitly for some primitive types). Decision variables are instantiated by the solving process, and their value for the current solution can be read.

3.4.1 Var types

OptimJ makes the difference between these two notions of variables by giving them different types. Decision variables have **var** types, identified by the keyword **var** followed by a type :

```
int x;
  // x is an imperative variable, ie. a location storing an integer

var int y;
  // y is a decision variable, whose solutions are integer values
```

Any type is allowed as the base type of decision variables in OptimJ, although the particular solver context may put restrictions on the set of allowed types.

```
class MyClass { }

abstract model VarTypes
{
      var float f;
      var double d;
      var int i;
      var boolean b;
      var String s; // needs a domain
      var MyClass m; // needs a domain
}
```

var types

(C) Ateji. All rights reserved.

Var types are the types of decision variables. They are written **var** T, where T is any Java type.

var is a type functional taking a type argument – it is not a declaration modifier. **var** is left-associative, meaning that **var int**`[]` is to be read as an array of **var int**, not as a var type ranging over **int**`[]`.

In the example above, OptimJ will complain that some variables need a domain : read on.

3.4.2 Declaring decision variables

Decision variables or arrays thereof must be declared as fields of a model (they cannot be declared as e.g. local variables of a method).

```
// OK: a decision variable in a field
var double d1;

// OK: an array of decision variables in a field
var double[] d2[10];

void aMethod()
{
      // Wrong : a local declaration
      var int d3;
}
```
decision variables must be fields of a model

A decision variable is always initialized. If you do not provide an initialization, OptimJ will do it for you if the variable is numeric or boolean.

You can think of decision variables as Java objects, which can be initialized with constructors of the appropriate type. The constructors will require you to provide a domain for each decision variable : the domain is the set of all possible solution values. For numeric types, OptimJ provides a familiar notation where you simply provide the bounds.

```
String[] stringDomain = { "abc", "def" };

// Java-style initialization
var int i1 = new var int(0,10);
var double d1 = new var double(0.0, 10.0);
var String s1 = new var String(stringDomain);

// modeling-style initialization - same as above
var int i2 in 0 .. 10;
var double d2 in 0.0 .. 10.0;
var String s2 in stringDomain;
```
initializing decision variables

The exact set of constructors available for a given type depends on the solver context. Refer to your solver context specification in the second part of this manual.

(C) Ateji. All rights reserved.

3.4.3 Provide tight bounds

If you do not provide the domain of a numeric variable, OptimJ will provide one for you. Non-numeric variables always require an explicit domain (there would be no way for OptimJ to guess the domain that you intended).

```
var int i; // in Integer.MIN_VALUE .. Integer.MAX_VALUE;
var double d; // in -Double.MAX_VALUE .. Double.MAX_VALUE;
var String s; // wrong : requires an explicit domain
```

implicit domains

Nevertheless, it is a good practice not to rely on implicit domains and always provide the tightest possible bounds for your variables. This can considerably speed-up running times, even more so for integral variables (whenever search is involved, the search space may grow exponentially with respect to the domain size). OptimJ will display a warning whenever you use an implicit (unbound) domain.

3.4.4 Declaring arrays of decision variables

You can also declare arrays of decision variables. Because Java-style declarations tend to make models unnecessarily verbose, OptimJ provides syntactic facilities known as *generalized declarations*.

In this exemple, we declare two arrays of ten decision variables and initialize them with their implicit domain :

```
// this generalized declaration
var int[] a1[10];

// is a short-hand for the following
var int[] a2 = new var int[10];
```

arrays of decision variables

Using associative arrays (see sectionErreur : source de la référence non trouvéeErreur : source de la référence non trouvéeErreur : source de la référence non trouvée for more details), you can index the array element using a string rather than a number:

```
var int[String] a3;
```

associative arrays of decision variables

Syntax note:

Notice that the array brackets must appear twice in a generalized declaration :

- once as a type (the type of a1 is an array of var int)
- once as a dimensioning expression (the size of a1 is 10)

Generalized array declarations can also provide domains. In this exemple we declare an array of ten decision variables whose domains are 0 .. 10 :

```
      var int[] a[10] in 0 .. 10;
```
domains in generalized declarations

Generalized declarations can also use generators. A generator introduces a name that ranges over the array elements, and this name can be reused in the domain expression.

In the following example, a is an array of 10 var int elements. The domain of a[0] is 0 .. 0, the domain of a[1] is 0 .. 2, the domain of a[2] is 0 .. 4, and so on. Similarly, b is a two-dimensional array of decision variables defined using two generators.

```
      var int[] a[int i : 10] in 0 .. 2*i; // i loops from 0 to 9
      var int[][] b[int i : 10][int j : 10] in 0 .. i+j;
```
generators in generalized declarations

3.4.5 Decision variables as Java objects

Depending on the context, decision variables can also be seen as Java objects. In the following example, the expression `v.m()` is in a constraints context, it refers to an invocation of the method `m()` in type **`C`**. (this means intuitively that in any solution, the value of `v` must be such that `v.m() == 0`):

```
abstract model M {
      var C v in CDomain;
      constraints {
            v.m() == 0;
      }
}
```
invoking a method on a decision variable

On the other hand, in a programming context, `v` is regarded as a plain Java objects having a specific reference type **`var C`**. You can do with `v` everything you can do with a plain Java object.

Remember however that objects of var types can only be declared inside a solver context, which basically means as fields of a *model* (they cannot be declared into a plain *class*).

3.5 Constraints

A constraint is any Java boolean expression involving decision variables. They have the same syntax as ordinary Java expressions. Constraints can be written within a **`constraints`** block and only there. A model may contain any number of constraints blocks.

(C) Ateji. All rights reserved.

The following are examples of constraints :

```
abstract model BasicConstraints
{
	var int i1, i2;
	var double d1, d2;

	constraints {
		2 * i1 != i2 + 3;
		Math.sin(d1) < d2*d2;
	}
}
```

basic constraints

The following is not a constraint since the expression does not involve decision variables. OptimJ will complain about a type error :

```
abstract model E12
{
	double alpha, beta;

	constraints {
		Math.sin(alpha) < beta*beta; // wrong
	}
}
```

not a constraint

When writing an abstract model, as we just did in the two previous examples, any Java expression is allowed as a constraint. Once you introduce a solver context for specifying a particular solver, only constraints that can be handled by this solver are allowed. Refer to your solver context specification in the second part of this manual.
A common requirement is that all constraints must be linear (decision variables can only be added, never multiplied). This is for instance the case of the `lpsolve` solver :

```
model Linear solver lpsolve
{
	var double d1, d2;
	constraints {
		2*d1 == 1 + 3*d2; // OK, linear
		d1 <= d2*d2; // Wrong, not linear
	}
}
```

linear constraints

3.5.1 Aggregate constraints

Aggregate operators such as sum, product, etc., allow expressions within constraints to range over arbitrary collections of data. They are the OptimJ equivalent of the well-known Σ-notation in mathematics.

As an example, the following constraint states that the sum of gains of four players must be zero :

```
constraints {
        sum{ int i : 0 .. 3 }{ gain[i] }  == 0;
}
```

a sum constraint

This is the same constraint as :

```
constraints {
        gain[0] + gain[1] + gain[2] + gain[3] == 0;
}
```

the same constraint fully developed

This sum example is a particular case of the more general concept of comprehension, which is detailed in section 7. The qualifier part can contain any number of generators and filters, the target part is any Java expression.

The aggregate operators allowed for constraints depend on the solver context. Refer to your solver context specification in the second part of this manual.

3.5.2 Collections of constraints, conditional constraints

Collections of constraints can be expressed with the **forall** construct, based on the same comprehension notation :

```
constraints {
        forall(Country c1 : cs, Country c2 : cs, :next(c1,c2)) {
                color[c1] != color[c2];
        }
}
```

a simple forall

`forall` constructions can be embedded and filters can be written as `if` clauses. This is the same as above:

```
constraints {
        forall(Country c1 : cs) {
                forall(Country c2 : cs) {
                        if(next(c1,c2)) {
                                color[c1] != color[c2];
                        }
                }
        }
}
```

embedded foralls and conditional constraints

Note that the `if` clause introducing a conditional constraint is unrelated to the `if` statement.

3.6 Solver-specific constraints

The OptimJ language itself is independant of any particular solver. You can however use specific constraints (typically the so-called "global constraints") of a given solver within a **constraints** block by prefixing them with the name of the solver context. Solver-specific constraints are described in each solver

(C) Ateji. All rights reserved.

context documentation in the second part of this manual. Obviously, models using solver specific constraints will not be portable across different solvers.

The following is an example of using a CPLEX specific SOS constraint. Note the `cplex11` prefix:

```
model M solver cplex11
{
    var int [] sosVars  [10] in -5 .. 5;
    double [] weights = new double [10];
    constraints {
        cplex11.SOS1(sosVars, weights);
    }
}
```

example of a solver-specific constraint

3.7 Objective

The objective declaration in a model is optional. You will write one when you want to minimize or maximize some expression (typically, you'll want to minimize a cost or maximize a profit). The expression after the **`minimize`** or **`maximize`** keyword is any Java numeric expression involving decision variables, exactly like a constraint.

For example, we may want to minimize the total cost of buying different foods :

```
      minimize sum{int j : nbFoods} { cost[j] * Buy[j] };
```

E17: objective function

3.8 Naming constraints and objective

Constraints and objective can be named. This is useful for getting information about the quality of your solution, such as dual values. Constraint names have the type **`constraints`**. You first declare the names, then associate them to your constraints or objective as follows :

```
      // declare the names
      constraints[] upperBound[nbFoods];

      constraints {
            forall(int j : nbFoods) {
                  // the j-th constraint is named upperBound[j]
                  upperBound[j]: Buy[j] <= max[j];
            }
      }
```

naming constraints

```
// declare the names
constraints totalCost;

// the objective is named totalCost
minimize totalCost: sum{int j : nbFoods} { cost[j] * Buy[j] };
```

naming the objective

3.9 The model life cycle

The following is a typical example of a model life cycle :

```
/*
 *  A typical application entry point.
 */
public static void main(String[] args)
{
        // instanciate the model
        MyModel myModel = new MyModel();

        // extract the model
        myModel.extract();

        // solve the model
        if(myModel.solve()) {
                // print the solutions
                // ...
        } else {
                System.out.println("No solution");
        }
}
```

the model life cycle

This example demonstrates the three phases of a model life cycle:

Instantiation

This is the same as instantiating a Java class : the model object is instantiated, the appropriate constructor and all the initializers are run, the solver is instanciated.

Extraction

The `extract()` method feeds the model to the solver.

Solving

The method `solve()` asks the solver to provide a solution. The solution values of the decision variables can be examined after each call to solve. Note that not all solver contexts provide a solve method: the file format contexts only extract a model into a file.

(C) Ateji. All rights reserved.

3.10 Getting solutions

A model predefines a number of methods. The first you will use is certainly `value()`, that returns the value of a decision variable after a successful `solve()` :

```
var double[] Buy[nbFoods];

public void printSolutions()
{
	for(int j : nbFoods) {
		System.out.println(value(Buy[j]));
	}
}
```

getting solutions

The set of predefined methods in a model depends on the solver context. Refer to your solver context specification in the second part of this manual.

3.11 Look at samples

Remember that OptimJ comes accompanied with a comprehensive library of code samples. They are a complement to this manual and can be a good starting point for writing your models.

(C) Ateji. All rights reserved.

4 Accessing the solver API

A major goal in the design of OptimJ is that the whole API of your solver must remain accessible.

The OptimJ language provides high-level notations for the concepts common to all solvers, such as decision variables, constraints and objective, in a natural and solver-independent way. But a single language cannot handle all the idiosyncracies of all solvers existing and to come. This is why OptimJ allows direct access to the solver API for solver-specific features.

4.1 Solver instance

When you `extract()` a model, OptimJ will create an instance of the solver associated to the solver context. By definition, the type of this instance depends on the solver context. The predefined model method `solver()` returns the solver instance associated to the model. With a solver instance at hand, you can now call all methods from your solver API.

The following example shows the model instance and its corresponding solver instance for the case of the `lpsolve` solver.

```
model MyModel solver lpsolve
{
      public static void main(String[] args)
      {
            // create a model instance
            MyModel myModel = new MyModel();

            // extract the model : this also instanciate a solver
            myModel.extract();

            // get the solver instance instanciated by extract()
            // its type depends on the solver context
            LpSolve mySolver = myModel.solver();

            // call solver-specific API methods on the solver
            // this may also throw solver-specific exceptions
            try {
                  mySolver.getLpName();
            } catch(LpSolveException e) {
                  System.out.println("Oops.");
            }
      }
}
```

model instance and solver instance

(C) Ateji. All rights reserved.

4.2 Variables and constraints instances

Depending on the solver context, variables and constraints may be stored internally as

- "extractable objects" (for instance, the ILOG Concert™ API)
- column or row number (all matrix-based linear solvers)
- or any other mean

OptimJ calls this internal representation the "backing object" of a variable or a constraint, although this backing object may not be an object in the Java sense. You will need to access this backing object in two situations :

- passing a backing object to a method call on the solver instance
- call API methods directly on the backing object (only possible when it is an actual Java object)

The backing objects of variables and constraints are dependent on the solver context, refer to your solver context specification in the second part of this manual.

The following example demonstrates the common case of matrix-based linear solvers. All such solvers provide an API based on a matrix representation of the model, where variables are identified by a column number and constraints by a row number.

We will solve a model with the lpsolve solver and ask for the dual values of variables and constraints. The column number for a variable is given by the model method column(). The row number for a constraint is the integer that has been used when naming the constraint (see section 3.8). The lpsolve API provides the getDualSolution() method, that returns an array with the dual values of all variables and constraints. In order to retrieve the dual of a given variable or constraint, you must provide its position in the array, which can be computed from its backing object (an integer).

```
var int someVariable;
constraints someConstraint;

constraints { someConstraint : someVariable != 0; }

void printDuals() throws LpSolveException
{
      // get a solver instance
      LpSolve lp = this.solver();

      // this is how values are stored in the duals array
      int size = 1 + lp.getNrows() + lp.getNcolumns();
      int firstRow = 1;
      int firstColumn = 1 + lp.getNrows();

      // get the duals array
      double [] duals = new double[size];
      lp.getDualSolution(duals);
```

(Continued on next page)

```
        // print the dual of someVariable
        System.out.println("dual of someVariable: " +
              reducedCost(someVariable));

        // print the dual of someConstraint
        int row = someConstraint;
        System.out.println("dual of someConstraint: " +
              duals[firstRow + row]);
    }
```

accessing the solver API

The above is just an example of how to use the solver API. Some common methods such as getting duals exist as model methods for most solver contexts, and they should be prefered since they do not introduce a dependency on a specific solver. In other words, the previous example should rather be written as follows:

```
    void printDuals()
    {
        // print the dual of someVariable
        System.out.println("dual of someVariable: " +
              reducedCost(someVariable));

        // print the dual of someConstraint
        System.out.println("dual of someConstraint: " +
              sensitivity(someConstraint));
    }
```

same example using model methods

The set of predefined methods depends on the solver context. Refer to your solver context specification in the second part of this manual.

5 Initialization

Because OptimJ is an extension of the Java language, it must follow the Java rules for initialization. These rules have some counter-intuitive implications that deserve a chapter of their own.

If you're not interested in gory details, skip this chapter and simply remember this rule, you will be safe:

Mark all fields **`final`**
(unless you expressly want them to be modifiable)

For instance:

```
var int x; // Bad !!!
final var int y; // Good !!!
```

mark all fields final

Yes, we didn't apply this rule in the examples of this manual. This is definitely bad practice, but we felt that learning is easier when you learn one thing at a time.

5.1 The Java final modifier

Marking a field as **`final`** means that the Java compiler will ensure that it will be properly initialized, and that its value will never change after initialization.

From a Java point of view, decision variables are fields refering to some fixed location in memory. This location will never change, hence it is always safe to mark decision variables as final.

5.2 Constructors

Java constructors instanciate objects, and the values given as parameter to a constructor can be used to initialize fields of the object.

Constructors are the recommended way to pass data to an OptimJ model. Avoid modifying shared fields from the outside, this is bad programming practice that leads to spaghetti code.

(C) Ateji. All rights reserved.

```
model MyModel solver lpsolve
{
	// lower and upper are parameters of the model
	double lower, upper;

	// their values are given in the constructor
	MyModel(double lower, double upper)
	{
		this.lower = lower;
		this.upper = upper;
	}

	var double x in lower .. upper;
}
```

parameter-passing via a constructor

5.3 Java initialization order

However, if you try to define an array whose size is given by a constructor parameters, the code may not behave as you expect. Consider the following example:

```
model MyModel solver lpsolve
{
	// size is a parameter of the model
	int size;

	// its value is given in the constructor
	MyModel(int size)
	{
		this.size = size;
	}

	// we want an array of the given size
	var double[] x[size];
}
```

wrong size for x !

The array x will always be instanciated with a size of zero ! What is happening ? If we add final modifiers to all fields, the compiler will tell us that something is wrong:

```
model MyModel solver lpsolve
{
	final int size;

	MyModel(int size)
	{
		this.size = size;
	}

	// Error: "The blank final field size
	// may not have been initialized"
	final var double[] x[size];
}
```

with final modifiers, the compiler signals an error

(C) Ateji. All rights reserved.

What is happening is that Java initializes objects in this order:

1. All the fields are initialized with the blank value for their type (0 for ints, null for references, etc.)
2. The superclass is initialized
3. The fields with an inline initialization are initialized.
4. The constructor code is run

In our example, the array `x` is initialized first (step 3), and only then the constructor code is executed (step 4). The value of `size` used during the initialization of `x` is thus the blank value for integers, namely zero.

If `size` if marked **`final`**, the compiler complains that something is wrong with the initialization. It does not complain if the **`final`** modifier is not present, and such errors are difficult to catch, this is why we recommend to mark all fields **`final`** as much as possible. This is anyway good programming practice, as it makes your code more understable and enables the compiler to perform some smart optimizations.

You can find all the details regarding initialization in the Java Language Specification available from http://java.sun.com/docs/books/jls/third_edition/html/j3TOC.html

5.4 Instanciate fields in a superclass

Marking all fields **`final`** will ensure that initialization problems are caught by the compiler, but what about the solution ?

As we have just seen, the only thing that gets executed before field initializations is the superclass initialization (step 2). We will thus move all problematic fields into a superclass.

Our example should thus be rewritten as:

```
model MyModel extends MyModelParams solver lpsolve
{
	MyModel(int size)
	{
		// first initialize the superclass
		super(size);
	}

	// the field size from the superclass is visible here,
	// with its correct initialized value
	final var double[] x[size];
}

class MyModelParams
{
	// field size has been moved to the superclass
	final int size;

	MyModelParams(int size)
	{
		this.size = size;
	}
}
```

problematic field moved into a superclass

(C) Ateji. All rights reserved.

6 Data modeling and bulk processing

Solving an optimization problem almost always involves a data processing phase before running the solver, in order to bring the data sources in a shape suitable for modeling.

OptimJ provides another way to store information: associative arrays.

OptimJ additionally provides tuples, since it is a very common form of data : in fact, tables in relational databases are sets of tuples.

OptimJ provides collections and aggregate operations that allow clean and concise code for bulk data processing.

6.1 Associative arrays

Erreur : source de la référence non trouvée

Associative arrays provide another way to store information, somewhere between maps and classical Java arrays. They behave exactly like Java arrays, but indices range over any given collection rather than over integers from 0 to n.

The indices must be given at array creation time and can never change (unlike a `Map`). Even if the collection used for defining the indices is modified later, the indices will not change.

Associative arrays are convenient for expressing optimization models in a concise mathematical-like fashion.

6.1.1 Associative array types

An associative array type is written by writing the type of the values followed by the type of the indices between brackets.

```
int[String] myAssociativeArray;
```

associative array types

Here **`int`** is the type of the values, and `String` the type of the indices. Associative arrays can also be multidimentional:

```
int[Month][Product] quantities;
```

multi-dimensional associative arrays

Associative and "classical" dimensions can be mixed freely. The following is an 3-dimensional array whose first index is associative (ranging over strings), the second index is classical (ranging over 0-based integers), and the third index is associative (ranging over booleans):

```
    int[String][][boolean] myAssociativeArray;
```

mixing associative and classical dimensions

6.1.2 Creating associative arrays

Both the keys and the values must be provided when creating an associative array. A specific initializer notation provides a convenient way to define associative arrays in-line:

```
    int[String] ages =
      { "Stephan" -> 37,
        "Lynda"   -> 29 };
```

associative array initialization

The initializer notation can also be used for multidimensional arrays:

```
    int[Month][Product] quantities =
      { car       -> {january  -> 12, february -> 17},
        motorbike -> {february -> 3,  october  -> 37} };
```

multi-dimensional associative array initialization

The **new** operator for classical arrays extends straightforwardly to associative arrays. The following instanciates an associative arrays whose indices are taken from the collection `names`, and all values are initialized to the default value of the value type (0 in this case):

```
    int[String] ages = new int[names];
```

associative array initialisation with default values

The generalized declarations syntax also applies to associative arrays:

```
  String[] names = {"Yoann", "Bill"}; // a collection of strings
  int[String] lengths[String name : names] = name.length();
  // lengths = { "Yoann" -> 5, "Bill" -> 4 };
```

associative array creation

6.1.3 Accessing associative arrays

The main operation defined on associative arrays is access to a value by a key:

(C) Ateji. All rights reserved.

```
ages["Stephan"]           // evaluates to 37
quantities[car][january]  // evaluates to 12
```

associative array access

You can iterate over all values as follows:

```
for(int age: ages)... //  iterate over all values
```

The field `keys` gives the array of indices that was used at array creation time:

```
for(String name: ages.keys)... //  iterate over all keys
```

Two other interesting operations exist: to know the number of keys present in an array, use field *length*. To check whether an array contains a key, use method *containsKey*.

```
ages.length                    // 2
ages.containsKey("Lynda")  // true
ages.containsKey("Peter")  // false
```

6.1.4 Pitfalls and usage patterns

Exactly like classical Java arrays, the types and the index values are written in reverse order:

```
// int is the base type
// int[Month] is an array of int indexed by Month
// int[Month][Product] is an array of int[Month]
//                      indexed by Product
int[Month][Product] quantities = ...;

// thus when you index the array quantities, you must provide
// first the Product, then the Month
quantities[motorbike][february]
```

indices types and values appear in reverse order

Like for classical Java arrays, it is quite common to forget what is the proper order of indices. With associative arrays, you can rely on the type-checker to guess what is the proper order:

```
quantities[february][motorbike]  // type error
```

indices types and values appear in reverse order

Note that an associative array whith indices ranging over a integer `Range` is not the same as a Java array.

```
int[Integer] a1[int i : 0 .. 9] = i;
// a1  ranges over 0-based integers but is not a Java array.
int[] a2[int i : 10] = i; // i ranges from 0 to 9.
// a2 is a java array.
int[Integer] a3[int i : 7 .. 13] = i;
// indexes of a3 are 7, 8, ..., 13
```

Associative arrays do incur a space overhead compared to classical Java arrays, with a factor of approximately 2 (for reference value types) or 4 (for primitive value types). Do not attempt to optimize your code too early: the additional type checking provided by associative arrays is helpful for catching common coding errors.

6.2 Tuples

OptimJ provides a notion of tuples similar to the one used in mathematics. Tuples are Java objects, with a specific tuple type.

6.2.1 Tuple types

A tuple type is written by enumerating the types of its components between two tuple markers. Here is a tuple type containing an integer and a string :

```
(: int, String :) myTuple;
```

tuple types

6.2.2 Tuple values

A similar notation is used for creating tuple values, with the keyword **new** :

```
myTuple = new (: 3, "Three" :);
```

tuple values

The only operation defined on tuples is access to a component by position :

(C) Ateji. All rights reserved.

```
int i = myTuple#0;     // i = 3
String s = myTuple#1; // s = "Three"
```

tuple access

The position is an integer literal (it cannot be an expression).

Tuples are immutable objects : once created, the values of the components cannot be changed. This enables OptimJ to « optimize away » tuple instantiation in many common cases.

A tuple can be seen as a class defined and instantiated on the fly, where indices replace field names.

A tuple being an object, it implements all the methods of `java.lang.Object`. Two tuples are **`equals()`** if their components are **`equals()`** side by side.

6.3 Collection aggregate operations

These operations provide the OptimJ equivalent of the mathematical set comprehension notation, that you use everytime you say something like "the set of ... such that ...". They build Java collections such as `HashSet` and `ArrayList`.

They are based on the powerful comprehension notation, which is detailed in section 7.

```
import java.util.ArrayList;
import java.util.HashSet;
import static com.ateji.monoid.Monoids.arrayList;
import static com.ateji.monoid.Monoids.hashSet;

      ...

            int[] a = { 1, 3, 2, 3, 2 };

            HashSet<Integer> s;
            ArrayList<Integer> l;

            s = `hashSet(){ 2*i | int i : a };
            display(s);

            l = `arrayList(){ 2*i | int i : a };
            display(l);
```

building collections with the comprehension notation

This example will display:

```
2 4 6
2 6 4 6 4
```

(C) Ateji. All rights reserved.

6.3.1 Use cases

Collection aggregate operations provide declarative abstractions quite similar to those used in mathematics. They are important to make your code readable: you could always replace them by a number of loop statements, but writing "the set of ... such that ..." carries much more meaning that writing a long sequence of imperative statements. Another benefit of using this declarative style is that it allows the compiler to perform high-level code optimizations.

A common problem when writing optimization models is that the input data is stored along a different "axis" then what is required by you model. For instance, countries and their respective areas may be provided as two arrays of same length:

```
import optimj.util.Range;
   ...
     Range indices = 0 .. 2;
     String[] names = { "Jamaica", "Japan", "Jordan" };
     double[] areas = { 10991.0, 377873.0, 89342.0 };
```

country names and areas as two arrays

On the other hand, your model may require that country names and areas are given as a list of tuples:

```
     ArrayList<(:String,double:)> countries;
```

country names and areas as a list of tuples

Transforming one representation into the other may involve about a dozen lines of not very meaningful Java code. Using OptimJ declarative notation, this is expressed in one line in an intuitive way:

```
countries =
  `arrayList(){ new (: names[i], areas[i] :) | int i : indices };
```

transforming a couple of arrays into a list of couples

Printing the elements of `countries` will display:

```
(: Jamaica ,10991.0 :) (: Japan ,377873.0 :) (: Jordan ,89342.0 :)
```

Use declarative notations whenever possible: your optimization code becomes more concise, more readable and maintainable, and is likely to be more efficient.

(C) Ateji. All rights reserved.

6.4 Sparse data

Often, when working with large models, the data for the model tends not to be dense. Sparse data typically involves multiple dimensions, but does not necessarily contain values for every combination of the indices.

Many storage formats (or data structures) have been proposed to represent sparse matrices. Most of them have a Java API and can be used with OptimJ.

We will focus on solutions using associative arrays and tuples. This allows to easily skip certain combinations of indices, that are not valid, by omitting them in the array.

6.4.1 Problem description

Consider the declaration:

```
Set<Warehouse> warehouses;
Set<Customer> customers;
int[Customer][Warehouse] transp[warehouses][customers];
```

Array transp may be sparse

transp is a two-dimensional associative array. It may represent the units shipped from a wharehouse to a customer. But a wharehouse delivers only to a subset of customers. Array *transp* may be sparse.

We will present two ways to exploit the sparsity: tuples and non rectangular associative arrays.

6.4.2 Formulation in OptimJ with a set of tuples

Let us declare a set of tuples that only contains the revelant pairs (warehouse, customer). It is used to defined the indices of associative array *transp*.

```
Set<(:Warehouse, Customer:)> routes = ...;
int[(:Warehouse, Customer:)] transp[routes]
```

Asociative array indexed by a set of tuples.

The sum of all units shipped is calculated as follow:

```
sum{ transp[route] | (:Warehouse,Customer:) route : routes };
```

Sum of all units shipped

(C) Ateji. All rights reserved.

6.4.3 Formulation in OptimJ with non rectangular associative arrays

Another solution is to use a two-dimensional non-rectangular arrays:

```
 Set<Warehouse> warehouses;
 Set<Customer> customers(Warehouse w) { return ...;}
int[Customer][Warehouse]
              transp[Warehouse w:warehouses][customers(w)];
```

Non-rectangular associative array

This declaration specifies an associative array *transp*. For a given Warehouse *w*, indices of *transp[w]* (computed by method *customers*) are a subset of set of all customers. *transp* is said to be non rectangular since two different warehouses may be associated to different sets of customers.

The sum of units shipped from a given wharehouse w is calculated as follow:

```
sum{ transp[w][c] | Customer c : transp[w] };
```

Sum of units shipped

6.5 Initialization

OptimJ extends the Java array initializer notation to all kinds of collections, tuples and objects.

```
String[] array = {"a", "b", "c"}; // Java array
```

the Java array initializer notation

Similarly, in OptimJ you can initialize collections by listing their values:

```
ArrayList<String> list = {"a", "b", "c"}; // any collection
```

collection initialization

You can initialize tuples by giving their components:

```
(: int, int :) point = {3, 2};
```

tuple initialization

The same notation can be used for any object: there must be exactly one constructor with the same number of parameters. Suppose that you have defined `MyClass` with a constructor `MyClass(int, double)` :

(C) Ateji. All rights reserved.

```
MyClass c = {1, 1.9};
// is equivalent to MyClass c = new MyClass(1, 1.9);
```

object initialization

If no appropriate constructor exists, or if there are multiple constructors with the same number of parameters, OptimJ will raise a type error.

The ambiguity between collection initialization and object initialization is resolved in favor of collections:

```
ArrayList<Integer> list = { 3 };
// this creates a one-element list containing the value 3
// if you want a list of length 3 with default values,
// use the explicit constructor new ArrayList<Integer>(3)
```

ambiguity

Initializers can be nested to any depth level, and can include associative arrays initializers:

```
// List of appointments for each customer.
ArrayList<Date>[String] calendar = {
  "Steve"  -> {{2008, 06, 10}, {2008, 06, 14}},
  "Monica" -> {{2008, 06, 11}},
  // ...
};
```

nested initializers

Remember that initialization are only valid in declaration statements:

```
ArrayList<String> list;
list = {"a", "b", "c"}; // wrong: this is an assignment,
                        // not an initialization
```

the initializer notation works only in an initialization

(C) Ateji. All rights reserved.

7 Comprehensions

7.1 Comprehension expressions

We have seen the comprehension notation appearing in many different constructions of the OptimJ language :

- in collection aggregate operations:

```
`hashSet(){ i | int i : 0 .. 10, :i%2==0 }
```

comprehension defining a collection

- in aggregate expressions:

```
sum{ int i : 0 .. 10, :i%2==0 }{ i }
```

comprehension defining a value

- in aggregate constraints :

```
forall(int i : 0 .. 10, :i%2==0) {
      a[i] != 0;
}
```

comprehension defining multiple constraints

- in generalized declarations (partially) :

```
var int[] a[int i : 0 .. 10] in -i .. i;
```

comprehension (generator only) defining an array

All these language constructs are based on the same notion of *comprehension*[1].

1 *See for instance : Leonidas Fegaras and David Maier. Towards an Effective Calculus for Object Query Languages. In Proceedings of the 1995 ACM SIGMOD Int'l Conference on Management of Data, May 1995.*

The general form of a comprehension is either

```
op { target | qualifier_list }
```

or

```
op { qualifier_list } { target }
```

These two forms are strictly equivalent, and choosing one or the other is mainly a matter of taste and tradition. The first one is reminiscent of the mathematical set comprehension notation (target first), while the second is more akin to the big-Σ notation (qualifiers first).

- **`op`** is an aggregate operator (see below for a list of all predefined operators)
- **`target`** is any expression
- **`qualifier_list`** is a comma-separated list containing any number of generators and filters :
  - a *generator* has the form **`type variable : expr`** , where
    - **`variable`** is the name of the new variable introduced by the generator
    - **`expr`** is an expression whose type implements the interface **`java.lang.Iterable`**, is an array type or is an integer type.
  - a *filter* has the form **`: expr`** , where **`expr`** is any boolean expression

Generators are used to iterate over a set of values:

```
sum { k | int k in 1 .. 10} // k loops from 1 to 10
sum { k | int k in 10} // k loops from 0 to 9
sum { k | int k in new int []{3,5}} // k takes value 3 and then 5
```

different types of generators

Generators and filters can appear in any number and any order. For instance, these three comprehension expressions are equivalent (as long as `f` does not have side effects):

```
int k;
k = sum{ int i : 10, int j : 1 .. 10, :i%2==0 }{ f(i,j) };
k = sum{ int i : 10, :i%2==0, int j : 1 .. 10 }{ f(i,j) };
k = sum{ int j : 1 .. 10, int i : 10, :i%2==0 }{ f(i,j) };
```

any number of generators and filters in any order

The scope of the variable introduced by a generators is all generators and filters appearing to its right, together with the target expression. A variable introduced by a generator cannot hide another local variable.

(C) Ateji. All rights reserved.

```
// Wrong; i is used before it is defined
k = sum{ int j : 0 .. 10, :i%2==0, int i : 0 .. 10 }{ f(i,j) };

// Wrong: k hides an existing local variable
k = sum{ int k : 0 .. 10 }{ k };
```
variable scopes within comprehensions

Comprehensions can be nested.

```
k = sum{ int i : 0 .. 10 }
        { sum{int j : 0 .. 10, :i%2==0 }{ f(i,j) } };
```
nested comprehensions

The type of a comprehension depends on the aggregate operator and on the type of the target expression.

7.2 Predefined aggregate operators

7.2.1 Collection operators

Collections aggregate operators build collections :

operator	*type of the target expression*	*type of the comprehension*	
`` `hashSet() ``	`T`	`HashSet<T>`	builds a set
`` `arrayList() ``	`T`	`ArrayList<T>`	builds a list

You will need to add the corresponding imports:

```
import static com.ateji.monoid.Monoid.*;
```

7.2.2 Primitive operators

The following operators are the aggregate version of associative binary Java operators, and have the same typing rules. For instance, `sum{ int i : 1 .. 4 }{ i }` is the same as `1 + 2 + 3 + 4`.

primitive aggregate operator	*corresponding binary operator*
sum	+
prod	*
or	|
and	&

Other aggregate operators map binary methods :

(C) Ateji. All rights reserved.

primitive aggregate operator	*corresponding binary method*
max	Math.max
min	Math.min

7.2.3 Constraint operators

There is exactly one constraint aggregate operator, **forall**, used to define collections of constraints.

(C) Ateji. All rights reserved.

8 OptimJ for Java developers

If you're a Java developer with little experience in optimization, this chapter is for you.

8.1 What is optimization ?

There exist many textbooks about optimization. An gentle introduction can be found on the web at http://en.wikipedia.org/wiki/Optimization_(mathematics).

8.1.1 Application areas

Optimization techniques are commonly used today in various areas, including :

- transportation
- electricity grids
- finance (portfolio optimization)
- network architecture
- shift planning
- equipment dimensioning
- frequency allocation
- freight loading
- database scheduling
- strategic pricing

and many other services you are using everyday.

8.2 Common classes of models and solvers

The "shape" of your model, and the kind of solver you use to solve it; can have a tremendous impact on execution time.

8.2.1 Linear models (LP) are generally safe

If your model is 100% linear, that is

- all your decision variables are of type float or double
- all your constraints are linear, ie. of the form a1*X1 + ... + an*Xn, or equivalent to this form

then you can expect any linear (LP) solver to find a solution reasonably fast, even with a large number of variables and constraints.

(C) Ateji. All rights reserved.

OptimJ can help you design a linear model : if the solver context you are using is linear, then OptimJ will ensure that your model is linear, and complain otherwise.

The point is that solving linear models involves almost no searching. Linear models allow for fast algorithms that directly jump to the solution, without much backtracking.

The only thing you have to be careful about is accuracy problems : basically, the values of all variables should be commensurate. If you happen to mix very small values (say, 10^{-100}) with very large values (say, 10^{+100}), you are set for numerical instabilities, and you may get some strange solutions.

Floating-point numbers can exhibit counter-intuitive behaviours if you don't know what they represent precisely. Before using floating-point numbers, be it in as an OptimJ decision variable or in any other Java program, you should make sure that you know what you're doing. A good starting point on the web is the Wikipedia article http://en.wikipedia.org/wiki/Floating_point

8.2.2 Quadratic models need quadratic solvers

Quadratic models are similar to linear models, but they allow multiplication of two decision variables in the objective function, in the constraints, or both.

8.2.3 MIP models often require some solver tweaking

A MIP model is a basically linear model, where some of the variables are required to take an integer (or set-valued) value.

They are much more difficult to solve than linear models because they involve search :

- Suppose you had a floating-point variable whose optimal solution is 0.5 : this suggests that the optimal integer value would be 0 or 1, but this is not always the case. The solver may need to explore many more alternatives.
- The search space can grow very quickly. If you only have 32 binary (valued 0 or 1) variables, the search space to explore is already 2^32, ie. all the possible values of an int.

You have probably heard the story of the indian priest who asked to his king to put a grain of wheat on the first square of a checker board, two grains on the second square, four grains on the third square, and so on : the total amount of wheat would be more than what the earth has ever produced : this is a typical case of combinatorial explosion, that you have to keep in mind whenever you design a model involving search.

Fortunately, solvers are clever enough to not explore the whole search space. This ability to cut down the search space is in fact one of the main difference between different solvers.

You can also help the solver. The difference between you, as the designer of the optimization model, and the solver is that you know the structure of the model you've designed. Most solvers provide an API that allow one to specify various parameters relevant to the search strategy, and a good set of parameters can make the difference between a program that solves your problem in 3 milliseconds or in 3 million years. All solvers have different parameters : refer to each solver's manual for details. A good solver is one that will provide a good set of default parameters for you.

Using OptimJ, you can experiment with many different solvers without ever having to rewrite your model : simply set the "solver" clause to the solver you want to use.

8.2.4 Other kinds of models require the appropriate solver

The OptimJ language does not put any kind of restriction on the kind of models you can write. When you design an abstract model, anything is allowed.

- expressivity : you need to provide a solver context that understand the constructions you have used in your model : if you write a trigonometric constraint (using e.g. sin() and cos() from the java.lang.Math package), then you need a solver that understands trigonometric constraints
- time : you need to provide a solver that behaves reasonably efficiently for the problem at hand, and if your problem is complex, to tweak a number of parameters in order to help the solver find a solution quickly

8.3 Some rules of thumb to remember

If you're new to optimization, the first thing to learn is that badly designed models may require millions of years before providing you with a solution. Don't panic ! Optimization techniques have been used for a long time and solve every day terribly difficult problems with great efficiency. The few guidelines we give here should help you designing good models.

8.3.1 Always provide tight bounds

You can tremendously help the solver by providing the tightest possible bounds for all your decision variables. Remember that if your model involves search (if you have integer or set-valued variables), the search space may possibly grow exponentially with respect to the size of the domain.

8.3.2 Remove symmetries

Often a problem has many "equivalent" solutions. Imagine that you have found one solution of a magic square:

2	7	6
9	5	1
4	3	8

Then all transformations of this solution by mirroring and rotation are also solutions of the magic square, for instance:

4	9	2
3	5	7
8	1	6

4	3	8
9	5	1
2	7	6

6	7	2
1	5	9
8	3	4

and many other... Mirroring and rotation are typical examples of symmetries.

Rather than let your solver work hard for enumerating all these equivalent solutions, add additional constraints that will ensure that your solver will never provide two or more equivalent solutions, for instance by requesting that the numbers in the grid must request a specific ordering. You can then enumerate the solutions yourself by exploring the symmetries. This may have an important impact on solving time.

9 OptimJ development environment

The OptimJ language is supported by a language specific development environment under Eclipse. This environment mimics the functionalities of the standard JDT (Java Development Toolkit) so that Java developers used to the JDT can switch to OptimJ with a minimal learning curve.

The OptimJ development environment features :

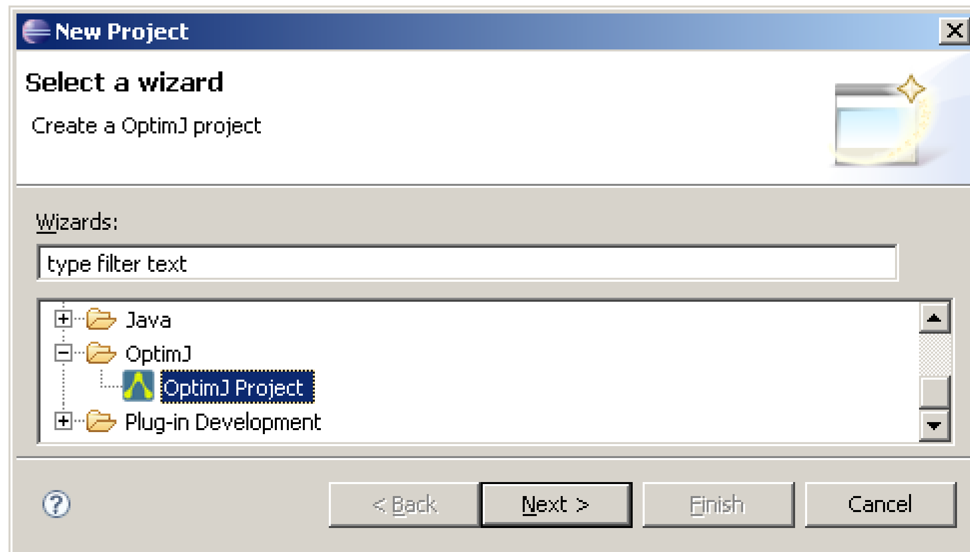
- a language-aware editor
- an OptimJ compiler
- navigation (in views : editor, outline, package explorer, type hierarchy)
- debugging (set breakpoints in the OptimJ editor, explore values in the debug perspective)
- launching

The OptimJ compiler is implemented as a source-to-source translator outputting pure Java source code. This approach has been chosen because it allows a smooth and easy integration with most Java-based programming tools, at the expense of a slightly longer compilation time :

- OptimJ and Java source files can coexist within the same project.
- All tools and plugins reading Java source code or byte code are directly usable with OptimJ programs.
- Tools that modify Java source code (code generators) are usually not compatible with OptimJ source files.

9.1 Create an OptimJ project

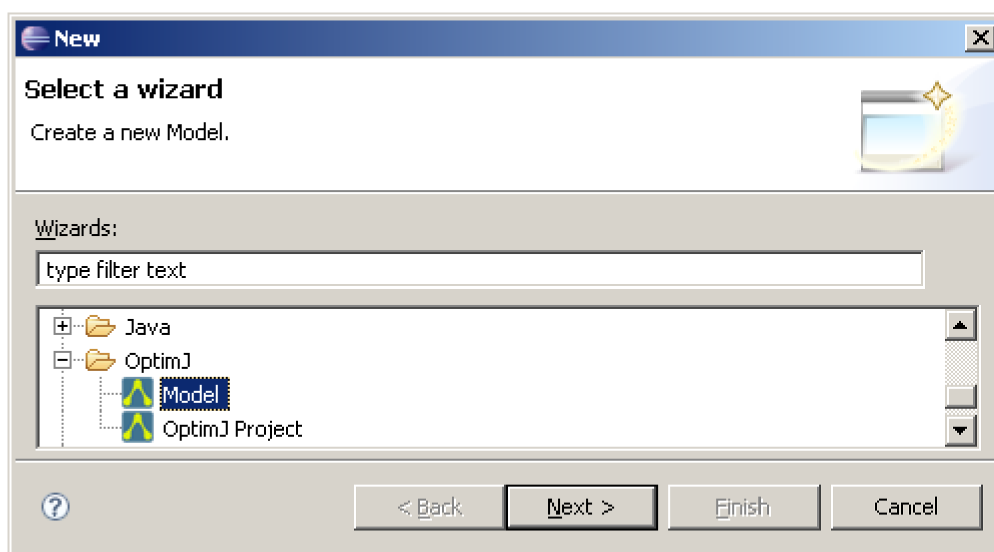
An OptimJ project is a Java project with an additional OptimJ nature. It can contain both Java and OptimJ source files. Select **File -> New -> Project...** and choose "OptimJ Project":



You can also convert an existing Java project into an OptimJ project (this will add the additional OptimJ nature). Right-click on the project name in the Package Explorer View and select:

You will notice that the icon of your project has changed to . Your project is ready to edit, compile and run OptimJ source files.

9.2 Create an OptimJ source file



Select "File" > "New" > "Other..." and choose Model dans la catégorie "OptimJ" -> "Model" :

(C) Ateji. All rights reserved.

This will create an OptimJ source file containing a skeleton of the model that you specified in the wizard.

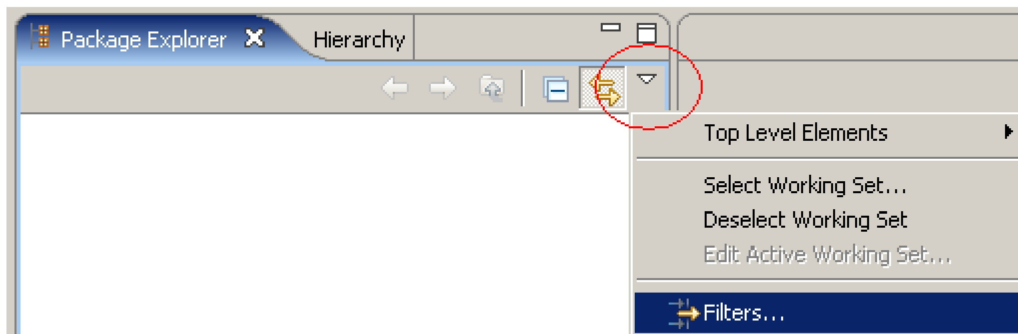
Alternatively, you can convert an existing Java file into an OptimJ source file. Right-click on the source file name in the Package Explorer View and select:

You will notice that the file icon has changed to and the editor background has become a light green. You can now start to write OptimJ code.

9.3 Show generated files

The OptimJ compiler does source-to-source translation behind the scenes. Every time your project is built, each OptimJ source file is translated into a Java source file of the same name.

You'd normally never need to look at the generated Java files, this is why they are hidden by default. If you want to see by yourself what happens under the hood, it is possible to deactive the corresponding filter. Open the Java Element Filters view:



Then select or deselect the Generated Java Files Filter:

9.4 Edit

An OptimJ editor has a light green background in order to visually distinguish from a Java editor :

It behaves like a Java editor except for the following :

- syntax coloring is specific to the OptimJ language
- refactoring actions (including quick fix and automatic imports) are not implemented

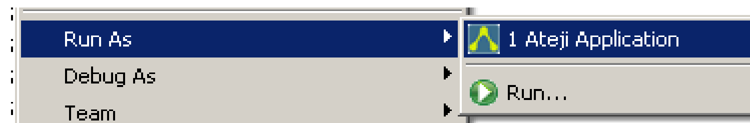
9.5 Compile

Like a Java source file, your code is compiled every time you save it. You can disable this behaviour by unchecking the "Build Automatically" option :

Project references and build path settings work as usual.

9.6 Run

If your OptimJ source file contains a main method, right-click on the file name in the Package Explorer view and select:



You can define run configurations and set run parameters as usual.

9.7 Debug

If your file contains a main method, start it with the "Debug as Ateji application" menu.:

If you have set a breakpoint in the OptimJ editor, the program will stop there.

(C) Ateji. All rights reserved.

Part II

Solver contexts

This part describes the solver contexts available for the OptimJ language. A solver context typically defines

- the types than can be used for decision variables
- the shapes allowed for constraints and the objective function
- the methods defined by the model

A solver context can be thought of as a "compile-time driver", it tells the OptimJ compiler how to generate code for a given solver API. In particular, solver contexts do not incur overhead at runtime.

A solver context is usually associated with a solver engine. Some contexts are used only to define the format of an output file for the OptimJ model, they are not associated with a solver and do not provide a `solve()` method.

Ateji is working hard to provide quickly solver contexts for the commonly available solvers. As a policy :

- Solver contexts will be provided freely for solvers available freely (other commercial solvers may require an additional licence)
- All new solvers contexts, including those requiring an additional licence, will be provided freely to existing customers with a maintenance contract

10 Matrix-based LP solvers

This section applies to all matrix-based linear solvers, ie. solvers where the model is defined as the coefficients of a large matrix. Most solvers in this class will have a similar solver context, this is why they are all grouped in a single chapter.

10.1 Common specifications

This section groups the specifications that are common to the following matrix-based linear solvers:

- `lpsolve`
- `glpk`
- `gurobi`
- `lp`
- `mps`

Specific details will be given in subsequent sections.

10.1.1 Variable types

The following types are allowed for decision variables:

- **`double`**
- **`int`**
- any reference type (expect subtypes of `java.lang.Number`)

10.1.2 Constraint shapes

Remember that a constraint is a boolean Java expression involving decision variables. The constraints handled by matrix-based solvers are basically first-order linear constraints:

- linear means that product of decision variables is prohibited
- first-order means that a constraint cannot contain another constraint

More formally, a constraint for a matrix-based solver context is a comparison of constraint terms, where a constraint term is:

- `X`, where `X` is a decision variable
- `X.f` or `X.m(...)`, where `X` is a decision variable over a reference type `T`, `f` is a field of `t` and `m(...)` is a method of `T`
- `a`, where `a` is a Java expression
- `a*X`, where `a` is a Java expression and `X` is a decision variable
- `X*b`, where b is a Java expression and `X` is a decision variable

(C) Ateji. All rights reserved.

- `a*X*b`, where a and b are a Java expressions and `X` is a decision variable
- `-t` or `+t`, where t is a constraint term
- `t1 + t2`, where `t1` and `t2` are constraint terms
- `sum{ ... }{ t }`, where t is a constraint term
- `(t)`, where `t` is a constraint term
- `a?t1:t2`, where `a` is Java expression and `t1, t2` are constraint terms

The comparisons handled by matrix-based solver contexts are boolean expressions built over terms `t1` and `t2` using the following operators, where at least one of `t1` or `t2` involves a decision variable:

- `t1 == t2`
- `t1 != t2`
- `t1 >= t2`
- `t1 <= t2`

The following are examples of valid constraints:

```
// a domain for String variables
String[] strings = { "abc", "def" };

// decision variables
var int I, J in 0 .. 10;
var int[] A[10] in 0 .. 10;
var double D in 0.0 .. 10.0;
var String S in strings;

// imperative variables
int i = 1;
double d = 1.0;
double f(int i, double d) { return i*d; }

constraints {
      2*I + 3 == 4*J + 5;
      2*(I+J) == 0;
      f(i,d) * I <= 0;
      f(i,d) * (I+J) >= 0;
      sum{int j : 0 .. 10}{f(j,d)*A[j]} != 0;

      S.charAt(0) <= 'a';
      S.length() >= 4;
}
```

sample linear constraints

The following are examples of invalid constraints:

(C) Ateji. All rights reserved.

```
constraints {
        I * J >= 1; // non linear
        (I >= J) == true; // higher-order
}
```

examples of constraints not handled by matrix-based LP solver contexts

10.1.3 Model methods

The following model methods are defined for all matrix-based LP solver contexts;

public void extract()

Extract the model into the solver.

public int column(var int)

public int column(var double)

public <T> int column(var T)

Return the column number of the given decision variable.

public int column(Constraints)

Return the row number of the given constraint.

public void dispose()

Free all memory allocated to the solver structure.

10.2 Lpsolve

`lpsolve` is OptimJ solver context associated to the lp_solve solver. For more information refer to the lp_solve project home page at http://lpsolve.sourceforge.net/5.5/.

10.2.1 Global constraints

`lpsolve` can handle SOS constraints written as follows:

```
addSOS(name, sostype, priority, count, sosvars, weights)
```

where

- `String name` is the name of the SOS constraint

(C) Ateji. All rights reserved.

- `int sostype` is the type of the SOS constraint (1 or 2).
- `int priority` is priority of the SOS constraint in the SOS set.
- `int count` is the number of variables in the SOS list.
- `var int[] sosvars` or `var double[] sosvars` or `var T sosvars` is an array specifying the variables.
- `double[] weights` is an array specifying the count variable weights.

10.2.2 Model methods

Solving the model:

`public boolean solve()`

Solve the model and returns true if lp_solve has found a feasible solution.

`extract()` must be called before calling `solve()`.

`solve()` can be called more than once. The model can be modified between calls to `solve()` using the lp_solve API.

`public int solverStatus()`

Return detailed information about the termination of the solver (see lp_solve documentation).

Getting the solution:

`public double objValue()`

Returns the objective value of solution.

`public public int value(var int)`

`public public double value(var double)`

`public public <T> T value(var T)`

`public int valueInt(var int)`

`public double valueDouble(var double)`

`public <T> T valueT(var T)`

Return the solution value of the variable.

(C) Ateji. All rights reserved.

public lpsolve.Lpsolve solver()

public double lowerBound(var double)

public double lowerBound(var int)

public <T> double lowerBound(var T)

public double lower (var double)

public double lower(var int)

public <T> double lower(var T)

Return the lower bound of the specified variable.

public double upperBound(var double)

public double upperBound(var int)

public <T> double upperBound(var T)

public double upper(var double)

public double upper(var int)

public <T> double upper(var T).

Return the upper bound of the specified variable.

public double reducedCost(var int)

public double reducedCost(var double)

public <T> double reducedCost(var T)

Return the reduced cost of the specified variable.

public double sensitivity(constraints)

Return the sensitivity of the specified constraint.

10.3 Glpk

`glpk` is OptimJ solver context associated to the glpk solver. For more information refer to the glpk project home page at http://www.gnu.org/software/glpk/.

10.3.1 Global constraints

There are no global constraints associated with GLPK.

(C) Ateji. All rights reserved.

10.3.2 Model methods

The list of methods is identical to that given above for lpsolve. The only difference concerns the solver method :

public org.gnu.glpk.GlpkSolver solver()

10.4 Gurobi

`gurobi` is the OptimJ solver context associated to the Gurobi solver. OptimJ is compatible with Gurobi versions 1 and 2. This section describes the solver context `gurobi` (associated to Gurobi 1.x) and `gurobi2` (associated to Gurobi 2.x). For more information refer to the Gurobi home page at http://www.gurobi.com/.

10.4.1 Global constraints

Gurobi can handle SOS constraints written as follows:

```
addSOS(vars, weights, type)
```

where

- `var int[] vars` is an array specifying the variables.
- `double[] weights` is an array specifying the count variable weights.
- `int type` is the type of the SOS constraint (1 or 2).

10.4.2 Model methods

The list of methods is identical to that given above for lpsolve. The only difference concerns the acces to the backing objects :

public gurobi.GRBModel solver()

public gurobi.GRBVar column(var int)

public gurobi.GRBVar column(var double)

public <T> gurobi.GRBVar column(var T)

public gurobi.GRBConstr row(constraints)

(C) Ateji. All rights reserved.

10.5 LP and MPS file formats

The lp and mps solver contexts are used to write OptimJ into text files in one of following standard formats LP or MPS.

Since they are not backed by any solver, they do not provide a solve() method, and none of the methods used to explore solutions. The only thing you can do with an lp or mps solver context is extract the model and output it to a standard text file. You can then feed this file to your favorite solver and solve the corresponding model using your solver API.

Here is a typical example of a model with an mps solver context:

```
model MPSModel solver mps
{
      // decision variables and constraints go here

      /*
       *  This method outputs the model into
       *  a standard text format. The FileWriter
       *  must be opened and closed by the caller.
       */
      static void writeModel(FileWriter out) throws IOException
      {
            // instanciate the model
            MPSModel myModel = new MPSModel();

            // extract it
            myModel.extract();

            // output it
            out.write(myModel.solver().toString());
      }
}
```

outputting a model into an MPS file

(C) Ateji. All rights reserved.

11 Mosek

`mosek` is the OptimJ solver context associated to the Mosek solver. For more information refer to the Mosek home page at http://www.mosek.com/.

11.1 Linear optimization

The `mosek` solver context handles all the common specifications for linear solver contexts as described in the previous section.

11.2 Quadratic optimization

In addition, it is possible to solve quadratic constrained convex problems using `mosek`. Here is an example:

```
constraints {
    1 <= x1 - x1*x1 - x2*x2 - 0.1*x3*x3 + 0.2*x1*x3;
}

minimize x1 * x1 + 0.1 * x2 * x2 + x3 * x3 - x1 * x3 - x2;
```

Quadratic optimization

In other words, we extend the shape of constraints with a new rule:

- `x * y` , where `x` and y are decision variables.

Note that x and y must be variables, not expressions. OptimJ will not expand products of expressions:

```
minimize (1 - x) * (1 - x); // Wrong. Expand your expression

minimize 1 - 2*x + x*x; // OK
```

No automatic expand

A very important restriction is that the problem should be convex: for instance, you can minimize `x*x` or maximize `-(x*x)`, but not the opposite. This property is checked at run-time by the solver. Refer to the Mosek documentation for details.

11.3 Conic optimization

Conic optimization is a particular case of quadratic optimization that can be solve much more efficiently. The `mosek` solver context can handle conic constraints written as follows:

- `appendcone(int conetype, double conepar, var int[] submem)`
- `appendcone(int conetype, double conepar, var double[] submem)`

with:

- `conetype` specifies the type of the cone. Two values are possible:
  - `mosek.Env.conetype.quad` (quadratic cone) or
  - `mosek.Env.conetype.rquad` (rotated quadratic cone).
- `conepar`: This argument is currently not used. Can be set to 0.0.
- `submem`: Variable subscripts of the members in the cone.

For example:

```
import mosek.Env;

constraints {
      // x4 >= sqrt{x0^2 + x2^2}
      mosek.appendcone(
            Env.conetype.quad, // quadratic cone
            0.0,
            new var double []{x[4],x[0],x[2]}
      );
}
```

Conic optimization

Refer to the Mosek documentation for more information about conic optimization.

11.4 Model methods

Solving the model:

public boolean solve()

Solve the model and returns true if mosek has found a feasible solution. If the problem is not convex, a solver exception is throwed.

`extract()` must be called before calling `solve()`.

`solve()` can be called more than once. The model can be modified between calls to `solve()` using the Mosek API.

public int solverStatus()

Return detailed information about the termination of the solver (see mosek documentation).

(C) Ateji. All rights reserved.

Getting the solution:

`public double objValue()`

Returns the objective value of solution.

`public int value(var int)`

`public double value(var double)`

`public <T> T value(var T)`

`public int valueInt(var int)`

`public double valueDouble(var double)`

`public <T> T valueT(var T)`

Return the solution value of the variable.

`public mosek.Task solver()`

`public double lowerBound(var double)`

`public double lowerBound(var int)`

`public <T> double lowerBound(var T)`

`public double lower (var double)`

`public double lower(var int)`

`public <T> double lower(var T)`

Return the lower bound of the specified variable.

`public double upperBound(var double)`

`public double upperBound(var int)`

`public <T> double upperBound(var T)`

`public double upper(var double)`

`public double upper(var int)`

`public <T> double upper(var T).`

Return the upper bound of the specified variable.

`public double reducedCost(var int)`

`public double reducedCost(var double)`

`public <T> double reducedCost(var T)`

Return the reduced cost of the specified variable.

`public double sensitivity(constraints)`

Return the sensitivity of the specified constraint.

(C) Ateji. All rights reserved.

12 CPLEX

OptimJ is compatible with CPLEX versions 9, 10, 11 and 12 available from www.ilog.com. This section describes the solver context `cplex9` and then presents additional features of `cplex10, cplex11 and cplex12`.

12.1 Common specifications

12.1.1 Variable types

The following types are allowed for decision variables:

- **`double`**
- **`int`**
- **`boolean`**
- any Java reference type (expect subtypes of `java.lang.Number`)

12.1.2 Constraint shapes

Remember that a constraint is a boolean Optimj expression involving decision variables.

Let us call S9 the set of OptimJ expressions that are valid constraints for the `cplex9` solver context. S9 is defined recursively as follows:

- `e`, where `e` is a decision variable
- `e.f` or `e.m(...)`, where `X` is a decision variable over a reference type `T`, `f` is a field of `t` and `m(...)` is a method of `T`
- `-e` or `+e`, where e belongs to S9.
- `e1 + e2`, where `e1` or `e2` belong to S9.
- `e1 * e2`, where `e1` or `e2` belong to S9.
- `sum{ ... }{ e }`, e belong to S9.
- `prod{ ... }{ e }`, e belong to S9.
- `(e)`, where `e belong to S9`
- `e1?e2:e3`, where e1 is an OptimJ boolean expression and e2 and e3 belong to S9.
- `?e`, where e belong to S9.
- `e1 == e2, where e1 or e2 belong to S9.`
- `e1 >= e2, where e1 or e2 belong to S9.`
- `e1 <= e2, where e1 or e2 belong to S9.`

`?e` is a new operator introduced in OptimJ for convenience: it transforms a boolean expression into a 0-1 integer expression.

(C) Ateji. All rights reserved.

The following are examples of valid constraints for `cplex9`:

```
// a domain for String variables
String[] strings = { "abc", "def" };

// decision variables
var int I, J in 0 .. 10;
var int[] A[10] in 0 .. 10;
var double D in 0.0 .. 10.0;
var String S in strings;

// imperative variables
int[] a = new int [10];
int i = 1;
double d = 1.0;
double f(int i, double d) { return i*d; }

constraints {
      2*I + 3 == 4*J + 5;
      (I+J) * 2 == 0;
      f(i,d) * I <= 0;
      f(i,d) * (I+J) >= 0;
      sum{int j : 0 .. 10}{f(j,d)*A[j]} == 0;

      I * J >= 1; // quadratic constraint

      S.charAt(0) <= 'a';
      S.length() >= 4;
}
```

valid cplex9 constraints

The following are examples of constraints that are not valid for `cplex9`:

```
constraints {
      (I >= J) | (I == 5); // higher-order
      I != J;
}
```

invalid cplex9 constraints.

12.1.3 CPLEX-specific constraints

The so-called global constraints of CPLEX can be used directly within an OptimJ **constraints** block by prefixing them with the name of the solver context. All global constraints presented here apply to `cplex10` and `cplex11`. Refer to the CPLEX documentation for the precise meaning of these constraints.

Square:

- `square(e)` where e is in S10/S11.

```
model M solver cplex11
{
    var int I in 0 .. 10, J in 5 .. 15;

    constraints {
        cplex11.square(I + J) >= 10; // (I+J)*(I+J) >= 10
    }
}
```

square

scalar product:

- `scalProd(vals, vars)`
- `scalProd(vals, vars,start,num)`

where

- `vars` is an array containing the variables the scalar product.
- `start` is the index of the first element to use in `vals` and `vars`.
- `num` is the number of elements to use in `vals` and `vars`.

```
model M solver cplex11
{
    var int [] A  [10] in -5 .. 5;
    int[] a = new int [10];
    constraints {
      cplex11.scalProd(a,A)  == 0; // a[0]*A[0]+...+a[9]*A[9]==0
      cplex11.scalProd(a,A,5,3)  == 0; // A[5]*a[5]+...+A[7]*a[7]==0
    }
}
```

scalProd.

special ordered sets

A special ordered set (SOS) constraint basically states that among a given collection of variables, only one variable (SOS of type 1) or only two consecutive variables (SOS of type 2) can be non-zero in any solution. Refer to the CPLEX documentation for details.

- `SOS1(vars, vals)`
- `SOS1(vals, vars)`
- `SOS1(vars, vals, start, num)`

- `SOS1(vals, vars, start, num)`

- `SOS2(vars, vals)`
- `SOS2(vals, vars)`
- `SOS2(vars, vals, start, num)`
- `SOS2(vals, vars, start, num)`

where

- `vars` - An array containing the variables in the new SOS.
- `vals` - An array containing the weight values for the variables in the new SOS.
- `start` - The first element in `var` and `val` to use for the new SOS.
- `num` - The number of elements in `var` and `val` to use for the new SOS.

Example:

```
model M solver cplex11
{
    var int [] sosVars  [10] in -5 .. 5;
    double [] weights = new double [10];
    constraints {
        cplex11.SOS1(sosVars, weights);
    }
}
```

Special Ordered Sets

piecewise linear function :

- `piecewiseLinear(expr,points,slopes,a, fa)`

- `piecewiseLinear(e,points,startPoints,num,slopes,startSlopes,a,fa)`

where

- `expr` - An expression indicating where to evaluate the piecewise linear function.
- `points` - An array containing breakpoints that define the piecewise linear function.
- `startPoints` - An integer indicating the first element in array `points` to use for the definition of the breakpoints of the piecewise linear function.
- `num` - The number of breakpoints to use from the array `points`. Thus `num+1` elements of array `slopes` are used.
- `slopes` - An array containing the slopes that define the piecewise linear function.
- `startSlopes` - The first element in array `slopes` to use for the definition of the slopes of the piecewise linear function.
- `a` - The first coordinate of the anchor point of the piecewise linear function.
- `fa` - The second coordinate of the anchor point of the piecewise linear function.

(C) Ateji. All rights reserved.

12.1.4 Model methods

The following model methods are defined for all versions of the `cplex` solver contexts;

`public void extract()`

Extract the model into the solver.

Solving the model:

`public boolean solve()`

Solve the model and returns true if cplex has found a feasible solution.

`extract()` must be called before calling `solve()`.

`solve()` can be called more than once. The model can be modified between calls to `solve()` using the cplex API.

`public ilog.cplex.IloCplex.CplexStatus solverStatus()`

Return detailed information about the termination of the solver (see cplex documentation).

Getting the solution:

`public double objValue()`

Returns the objective value of solution.

`public int value(var int)`

`public double value(var double)`

`public <T> T value(var T)`

`public int valueInt(var int)`

`public double valueDouble(var double)`

`public <T> T valueT(var T)`

Return the solution value of the variable.

`public void dispose()`

Free all memory allocated to the solver structure.

`public IloNumVar toConcert(var int)`

`public IloNumVar toConcert(var double)`

(C) Ateji. All rights reserved.

public <T> IloNumVar toConcert(var T)

Return the IloNumVar of the given decision variable.

public IloConstraint toConcert(constraints)

Return the IloConstraint of the given constraint.

public ilog.cplex.IloCplex solver()

Return the model instance.

public double lowerBound(var double)

public double lowerBound(var int)

public <T> double lowerBound(var T)

public double lower (var double)

public double lower(var int)

public <T> double lower(var T)

Return the lower bound of the specified variable.

public double upperBound(var double)

public double upperBound(var int)

public <T> double upperBound(var T)

public double upper(var double)

public double upper(var int)

public <T> double upper(var T).

Return the upper bound of the specified variable.

public double reducedCost(var int)

public double reducedCost(var double)

public <T> double reducedCost(var T)

Return the reduced cost of the specified variable.

public double sensitivity(constraints)

Return the sensitivity of the specified constraint.

12.2 Cplex10 and Cplex11

Solver contexts `cplex10` and `cplex11` extend `cplex9` with new constraints and new features.

(C) Ateji. All rights reserved.

12.2.1 Constraint shapes

Let S10 be set of OptimJ expressions that are valid constraints for `cplex10`. S10 is defined recursively as follows:

- If `e` belongs to S9 then `e` belongs to S10.
- `java.lang.Math.abs(e)` where `e` belongs to S10.
- `java.lang.Math.min(e)` where `e` belongs to S10.
- `java.lang.Math.max(e)` where `e` belongs to S10.
- `e1 | e2`, where `e1` and `e2` belong to S10.
- `e1 & e2`, where `e1` and `e2` belong to S10.
- `or{ ... }{ e },` where `e` belongs to S10.
- `and{ ... }{ e }`, where `e` belongs to S10.
- `min{ ... }{ e }`, where `e` belongs to S10.
- `max{ ... }{ e }`, where `e` belongs to S10.
- `e1 => e2`, where `e1` and `e2` belongs to S10.
- `e1 != e2`, where `e1` or `e2` belongs to S10.
- `!e`, where `e` belongs to S10.

The set S11 of OptimJ expressions that are valid constraints for the `cplex11` solver context is the same as S10.

The following are examples of valid constraints for `cplex10` and `cplex11`:

```
// decision variables
var int X in 0 .. 10;
var int Y in 0 .. 10;
var int[] A[10] in 0 .. 10;

constraints {
      (X == 0) | (X == 1);
      (X == 0) => (Y != 0);
      // if (X==0) is true then (Y!=0) must also be true
      max{var int Ai : A}{Ai} != 10;
      java.lang.Math.abs( X - Y) >= 5;
      !(cplex10.square(X - Y) <= 3);
      sum{var int Ai : A}{?(Ai == 0)} == 3;
      // There exists three i s.t. Ai == 0.
}
```

valid cplex10/cplex11 constraints

13 Table of Contents

1 Features and Benefits....3
2 OptimJ Jump Start....4
3 Language constructs for modeling....8
3.1 The basics : optimization concepts in Java....8
3.2 Solver independence and solver integration....9
3.3 Solver context....10
3.4 Decision variables....11
3.4.1 Var types....11
3.4.2 Declaring decision variables....12
3.4.3 Provide tight bounds....13
3.4.4 Declaring arrays of decision variables....13
3.4.5 Decision variables as Java objects....14
3.5 Constraints....14
3.5.1 Aggregate constraints....15
3.5.2 Collections of constraints, conditional constraints....16
3.6 Solver-specific constraints....16
3.7 Objective....17
3.8 Naming constraints and objective....17
3.9 The model life cycle....18
Instantiation....18
Extraction....18
Solving....18
3.10 Getting solutions....19
3.11 Look at samples....19
4 Accessing the solver API....20
4.1 Solver instance....20
4.2 Variables and constraints instances....21
5 Initialization....23
5.1 The Java final modifier....23
5.2 Constructors....23
5.3 Java initialization order....24
5.4 Instanciate fields in a superclass....25
6 Data modeling and bulk processing....26
6.1 Associative arrays....26
6.1.1 Associative array types....26
6.1.2 Creating associative arrays....27
6.1.3 Accessing associative arrays....27
6.1.4 Pitfalls and usage patterns....28
6.2 Tuples....29
6.2.1 Tuple types....29
6.2.2 Tuple values....29
6.3 Collection aggregate operations....30
6.3.1 Use cases....31
6.4 Sparse data....32
6.4.1 Problem description....32
6.4.2 Formulation in OptimJ with a set of tuples....32
6.4.3 Formulation in OptimJ with non rectangular associative arrays....33
6.5 Initialization....33

(C) Ateji. All rights reserved.

7 Comprehensions....35
7.1 Comprehension expressions....35
7.2 Predefined aggregate operators....37
7.2.1 Collection operators....37
7.2.2 Primitive operators....37
7.2.3 Constraint operators....38
8 OptimJ for Java developers....39
8.1 What is optimization ?....39
8.1.1 Application areas....39
8.2 Common classes of models and solvers....39
8.2.1 Linear models (LP) are generally safe....39
8.2.2 Quadratic models need quadratic solvers....40
8.2.3 MIP models often require some solver tweaking....40
8.2.4 Other kinds of models require the appropriate solver....41
8.3 Some rules of thumb to remember....41
8.3.1 Always provide tight bounds....41
8.3.2 Remove symmetries....41
9 OptimJ development environment....42
9.1 Create an OptimJ project....43
9.2 Create an OptimJ source file....43
9.3 Show generated files....44
9.4 Edit....44
9.5 Compile....45
9.6 Run....45
9.7 Debug....45
10 Matrix-based LP solvers....47
10.1 Common specifications....47
10.1.1 Variable types....47
10.1.2 Constraint shapes....47
10.1.3 Model methods....49
10.2 Lpsolve....49
10.2.1 Global constraints....49
10.2.2 Model methods....50
10.3 Glpk....51
10.3.1 Global constraints....51
10.3.2 Model methods....52
10.4 Gurobi....52
10.4.1 Global constraints....52
10.4.2 Model methods....52
10.5 LP and MPS file formats....53
11 Mosek....54
11.1 Linear optimization....54
11.2 Quadratic optimization....54
11.3 Conic optimization....55
11.4 Model methods....55
12 CPLEX....58
12.1 Common specifications....58
12.1.1 Variable types....58
12.1.2 Constraint shapes....58
12.1.3 CPLEX-specific constraints....59
12.1.4 Model methods....62
12.2 Cplex10 and Cplex11....63
12.2.1 Constraint shapes....64
13 Table of Contents....65

(C) Ateji. All rights reserved.

(C) Ateji. All rights reserved.