JSR-331

Java Constraint Programming API

SPECIFICATION

Version:

0.7.1

Status:

Public Draft/Final Release

Specification Lead:

Jacob Feldman, Cork Constraint Computation Centre

Java Community Process

www.jcp.org

October-2010

Table of Contents

INTRODUCTION

This JSR-331 is a Java Specification Request being developed under the Java Community Process rules. This specification defines a Java runtime API for Constraint Programming.

Rationale

Constraint Programming (CP) is a programming paradigm which provides useful tools to model and efficiently solve constraint satisfaction and optimization problems. Today CP is a proven optimization technique and many CP solvers empower real-world business applications in such areas as scheduling, planning, configuration, resource allocation, and real-time decision support. However, the absence of standards still limits the acceptance of CP by the business world.

This specification addresses the need to reduce the cost associated with incorporating constraint-based engines within business applications and the need to reduce the cost associated with implementing platform-level optimization tools and services. A number of vendor-specific CP APIs do exist. However, the differences between these specifications are significant enough to cause costly difficulties for application developers, platform vendors, researchers, and software architects.

Objective

The standardization of Constraint Programming aims to make CP technology more accessible for business software developers. Having a unified Java interface will allow commercial Java application developers to model their problems in such a way that the same model can be tried with different CP solvers. This will minimize vendor dependence without limiting vendor's innovation. At the same time, the standardized API will help to bring the latest research results to real-world business applications.

The objectives of the specification are to:

- Facilitate adding constraint-based decision support technology to Java applications
- Increase communication and standardization between CP vendors.
- Encourage the creation of a market for vertical constraint-based application and tool vendors through a standard CP API
- Facilitate embedding constraint-based techniques in other JSRs to support declarative programming models
- Make Java applications more portable from one constraint solver vendor to another.
- Provide implementation patterns and supporting libraries of constraints and search strategies for different constraint-based applications

- Support CP vendors by offering a harmonized API that meets the needs of their existing customers and is easily implemented.

Target Audience

The specification is aimed at three major audiences:

1) Business application developers who will use the CP API to develop real-world decision support application using the standard, vendor-neutral, Java interface
2) Constraint solver vendors who will develop and maintain their own implementations of the CP API compliant with the specification
3) CP Researchers, who will provide and enrich libraries of the standard constraints, search algorithms, and concrete problems that will be maintained by the CP Community.

Scope

The scope of the specification is to define an easy-to-use, lightweight-programming interface that constitutes a standard API for acquiring and using a constraint solver.

The specification targets the Java-based platforms and is compatible with JDK1.5 or higher.

The scope of the specification follows a minimalistic approach with emphasis on ease to use. It covers commonly accepted CP concepts and their representations that have been already de-facto standardized in various CP solvers and scientific articles. At the same time, the scope of the specification is broad enough to allow business application developers to use the standardized CP interfaces to model and solve typical constraint satisfaction problems in the most popular business domains such as scheduling, resource allocation, and configuration.

The following concepts and items are in the initial scope of the specification:

- Constrained variables of major types Integer, Boolean, Real, and Set
- Unary and binary constraints and constrained expressions defined on constrained variables
- The most popular global constraints
- An ability to find a solution, all solutions, and optimal solutions under certain, user-defined, limits.

The CP API is expected to be extended and thus it also specifies how to add new CP concepts, features, and strategies as they become commonly accepted.

The JSR-331 concentrates only on the CP interface for application developers and does not assume any particular implementation approach. The following concepts and items

are outside the scope of the specification and remain a prerogative of different CP solvers and specialized CP tools:

- Domain implementation mechanisms for different domain types
- Implementations of major binary and global constraints
- Constraint propagation mechanisms
- Backtracking and reversibility mechanisms.

Compliance

Compliance is of interest to the following audiences:

- Those designing, implementing, or maintaining JSR-331 implementations
- Governmental or commercial entities wishing to procure JSR-331 implementations
- Testing organizations wishing to provide a JSR-331 compliance test suite
- Software developers wishing to use the same code for problem definition and problem resolution when they decide to switch between different but compliant JSR-331 implementations
- Researchers wishing to demonstrate, and make freely available for testing and actual use the results of their research
- Educators wishing to teach JSR-331 compliant constraint programming courses
- Authors wanting to write about JSR-331 compliant CP solvers.

The text in this specification that specifies requirements is considered normative. All other text in this specification is informative, that is, for information purposes only. Normative text is further broken into required and conditional categories. Conditionally normative text specifies requirements for a feature such that if that feature is provided, its syntax and semantics must be exactly as specified. If any requirement of the specification is violated, the behavior is undefined.

Document Conventions

The regular Century Schoolbook font is used for information that is prescriptive by this specification.

The italic Century Schoolbook font is used for paragraphs that contain descriptive information, such as notes describing typical use, or notes clarifying the text with prescriptive specification.

[While the document is under review, we will use italic blue font to write notes and questions for reviewers that should not be considered as a part of the specification. The reviewer notes will be taken in brackets]

`The Courier New font is used for code examples.`

Within this specification the words “should” and “must” mean that compliant implementations are required to behave as described.

All examples in this specification are for illustrative purposes and are non-normative. If an example conflicts with the normative prose the prose always takes precedence. A significant portion of the specification resides in the documentation of the API (javadoc). The javadoc is normative.

ARCHITECTURE

The JSR-331 prescribes a set of fundamental operations used to define and solve constraint satisfaction and optimization problems. The JSR-331 consists of three major components:

- Specification (CP API)
- JSR-331 Implementations based on different CP solvers
- Technology Compatibility Kit (TCK) - a suite of tests, tools, and documentation that will be used to test implementations for compliance with the specification.

Architecturally the JSR-331 can be viewed as:

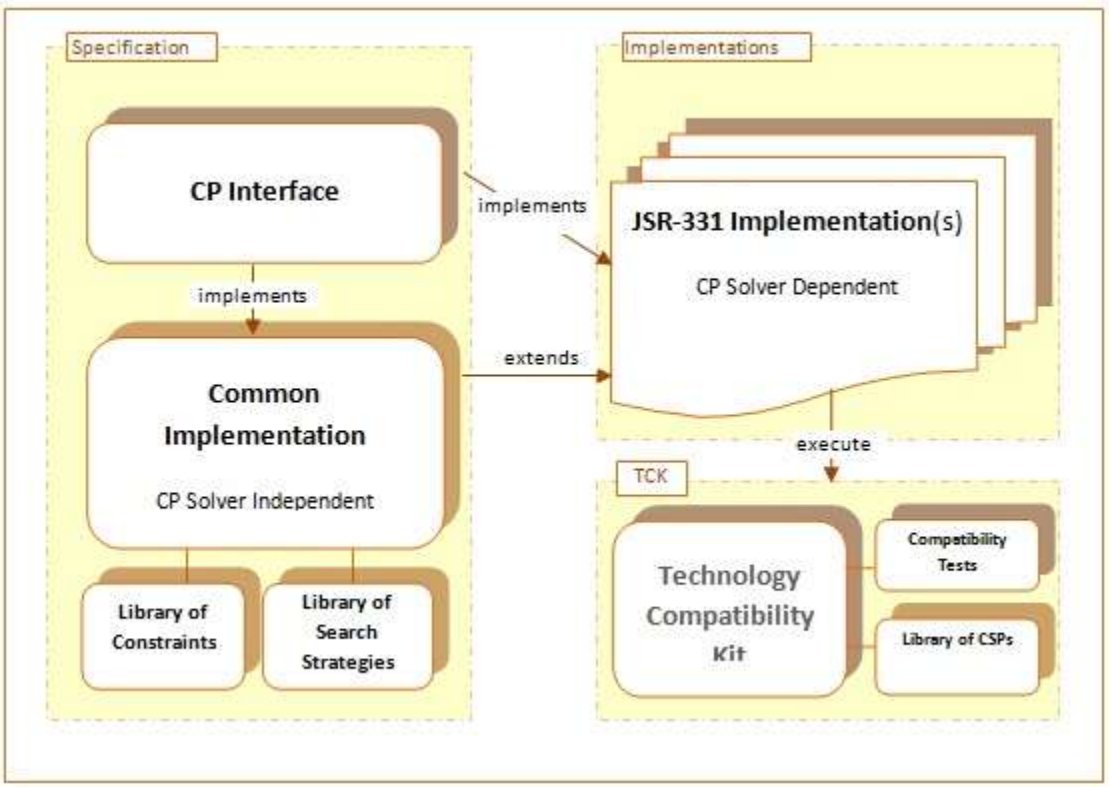

Figure 1. JSR-331 Architecture

Every JSR-331 Implementation may implement the CP Interface directly or by extending the Common Implementation classes. The Common Implementation may essentially simplify JSR-331 implementation efforts. As the standard becomes more mature and many implementations have more common constraints and search strategies, they will be gradually added to the Common Implementation libraries.

Specification

The JSR-331 specification consists of:

- Pure CP Interface (`javax.constraints`)
- Common Implementation (`javax.constraints.impl`).

The specification is completely independent of its implementations and defines a standardized CP API using the following packages:

Java Package	Description
`javax.constraints`	Pure Java interfaces such as Problem and Solver that specify major concepts and methods to define and solve constraint satisfaction and optimization problems
`javax.constraints.impl`	Java classes such as ProblemI and SolverI that implement (partially or completely) problem definition and resolution concepts and methods that do not depend on concrete CP solvers
`javax.constraints.impl.constraint`	This is a "Library of Constraints" that contains implementations of commonly used constraints which are not aware of the specifics of concrete implementations (CP solvers)
`javax.constraints.impl.search`	This is a "Library of Search Strategies" that contains implementations of commonly used search strategies which are not aware of the specifics of concrete JSR-331 implementations
`javax.constraints.impl.search.selector`	Implementations of commonly used selectors for variables inside an array of variables (e.g. VarSelectorMinDomain) and values inside variable domains (e.g. ValueSelectorMaxDegree). These implementations are not aware of the specifics of concrete JSR-311 implementations and can be used by different search strategies

JSR-331 Implementations

Any implementation of the JSR-331 specification is based on a concrete CP solver and provides implementation classes for all interfaces defined in the package "***javax.constraints***". Some implementation classes can directly implement the standard interfaces and others can be inherited from common implementations provided in the specification package "***javax.constraints.impl***". The JSR-331 requires any implementation to provide as a minimum a strictly defined list of implementation classes within the following Java packages:

Java Package	Description
javax.constraints.impl	Java classes such as Problem and Solver that provide final implementations for problem definition and resolution concepts and methods. These implementation classes can be inherited from the common (usually abstract) classes defined in the package with the same name but on the specification level, e.g. class `javax.constraints.impl.Problem` extends `javax.constraints.impl.ProblemI` that implements `javax.constraints.Problem`
javax.constraints.impl.constraint	This is a "Library of Constraints" that contains implementations of basic and global constraints which are based on concrete CP solvers
javax.constraints.impl.search	This is a "Library of Search Strategies" that contains implementations of search strategies which are based on concrete CP solvers.

Additionally every implementation may also provide its own ("native") constraints and search strategies assuming that they follow the standardized interfaces `javax.constraints.Constraint` and `javax.constraints.SearchStrategy`.

The fact that all JSR-331 implementations will use the same names for packages, major classes and methods will allow business application developers to easily switch between different implementations *without any changes* in the application code. They can write application-specific constraint-based engines once using only common CP API and use different CP solvers by changing only implementation-specific jar-files in their classpath.

Note. An ability to switch between underlying solvers with "no changes in the application code" comes with a price: the fixed naming convention for implementation packages means that JSR-331 based applications cannot mix two different implementations at the same

application code. The choice of an underlying implementation is defined only by a jar file in the application classpath.

Technology Compatibility Kit

The Technology Compatibility Kit (TCK) is a suite of tests, tools, and documentation that will be used to test implementations for compliance with the JSR-331 specification. The TCK is based on the JUnit framework and includes JUnit tests that cover all mandatory features included into the latest release of the JSR-331.

The JSR-331 TCK consists of two packages:

Java Package	Description
`org.jcp.jsr331.tests`	This package contains JUnit modules that allow a user to automatically validate if a concrete implementation is compliant with the JSR-331 (i.e. produces expected results for major functional tests)
`org.jcp.jsr331.samples`	This package contains CSP samples that provide integrated tests for major CP constraints and search strategies included in the JSR-331 and demonstrates the use of CP for real-world problems

Not all concepts introduced in the package "`javax.constraints`" are required to be implemented by a CP solver to be compliant with the JSR-331. The package "`org.jcp.jsr331.tests`" includes only those tests that are normative (must be satisfied by any implementation for compliance purposes). The module "`AllTests.java`" inside this package lists all normative tests.

The package "`javax.constraints.impl`" provides default implementations for some optional interfaces and methods. There are two types of the optional implementations:

1. Actual default implementations (not always the most efficient) that can be overridden by a particular JSR-331 implementation
2. Simple stubs that throw runtime exceptions informing a user that these methods are not implemented by a particular implementation.

Note. The package "`org.jcp.jsr331.samples`" is for demonstration purposes only and not all included samples have to be supported by all implementations.

Deployment Model

The deployed business applications that utilize the JSR-331 API will require the following jar-files in its classpath:

- `jsr331.jar`: includes all standard specification interfaces and classes
- `jsr331.<solver>.jar`: includes all implementation specific classes
- `<solver>.jar`: include all classes for the CP solver based on which this implementation was created.

For example, a Choco-based deployment will need the following jars:

- `jsr331.jar`
- `jsr331.choco.jar`
- `choco.jar`

Note. The JSR-331 does not depend on a particular implementation of logging mechanisms and does not need logging jars. However, all JSR-331 implementations must provide their own logging by implementing only basic methods `log`(string), `debug`(string), and `error`(string) inside the class Problem.

Constraint Satisfaction Problem (CSP)

Many real-life problems that deal with multiple alternatives can be presented as constraint satisfaction problems (CSP) and can be successfully solved by applying different Constraint Programming tools.

Formal Definition

Formally a Constraint Satisfaction Problem is defined by

a set of **variables** V1, V2, … Vn, and

a set of **constraints**, C1, C2, … Cm.

Each variable Vi has a non-empty **domain** Di of possible **values**. Each constraint Cj involves some subset of the variables and specifies the allowable combinations of values for that subset. A state of the problem is defined by an **assignment** of values to some or all of the variables. A **solution** to a CSP is an assignment that satisfies all the constraints. If a CSP requires a solution that maximizes or minimizes an objective function it is called “constraint optimization problem”. We will use an abbreviation CSP for both types of problems.

The main CSP search technique interleaves various forms of search with constraint propagation, in which infeasible values are removed from the domains of the variables through reasoning about the constraints.

Major JSR-331 Concepts

JSR-331 defines all necessary Java concepts to allow a user to represent and solve different Constraint Satisfaction Problems. JSR-331 supports a clear demarcation between two different CSP parts:

1) **Problem Definition** represented by the interface Problem

2) **Problem Resolution** represented by the interface Solver.

Correspondingly, all major CP concepts belong to one of these two categories. At the very high level a business user is presented with only 6 major concepts:

Problem
- Constrained Variable
- Constraint

Solver
- Search Strategy
- Solution

While different CP solvers use different names and representations for these major concepts, semantically these 6 concepts are invariants for the most of them. JSR-331 provides a unified naming convention and detailed specifications for these concepts.

The Problem Definition does not "know anything" about the Problem Resolution. An instance of the class Problem may exists without any Solver being created. Contrary, an instance of the class Solver may be created only base on a particular problem. During solution search, a solver can change a problem state (such as domains of constrained variables). It is the responsibility of a particular solver to keep (or not) different problem states based on the selected search strategy it defines.

Introductory Example

The following example demonstrates how a problem definition and a problem resolution are presented using JSR-331 API for a very simple arithmetic problem:

```
import javax.constraints.impl.Problem;
import javax.constraints.Var;
import javax.constraints.Solver;
import javax.constraints.Solution;

public class Test extends Problem {

  public void define() { // PROBLEM DEFINITION
   //======= Define variables
   Var x = variable("X",1,10);
   Var y = variable("Y",1,10);
   Var z = variable("Z",1,10);
   Var r = variable("R",1,10);
   Var[] vars = { x, y, z, r };
   //======= Define and post constraints
   try {
         post(x,"<",y);           // X < Y
         post(z,">",4);           // Z > 4
         post(x.plus(y),"=",z);   // X + Y = Z
```

```
            postAllDifferent(vars);
            int[] coef1 = { 3, 4, -5, 2 };
            post(coef1,vars,">",0); // 3x + 4y -5z + 2r > 0

            post(vars,">=",15);     // x + y + z + r >= 15


            int[] coef2 = { 2, -4, 5, -1 };
            post(coef2,vars,">",x.multiply(y));// 2x-4y+5z-r > x*y

      } catch (Exception e) {
            log("Error posting constraints: " + e);
            System.exit(-1);
      }
   }

   public void solve() { // PROBLEM RESOLUTION
      log("=== Find Solution:");
      Solver solver = getSolver();
      Solution solution = solver.findSolution();
      if (solution != null)
            solution.log();
      else
            log("No Solution");
      solver.logStats();
   }

   public static void main(String[] args) {
      Test p = new Test();
      p.define();
      p.solve();
   }
}
```

This code will produce the results that may look like below:

```
=== Find Solution:
Solution #1:
        X[1] Y[4] Z[5] R[6]
*** Execution Profile ***
Number of Choice Points: 3
Number of Failures: 1
Occupied memory: 4503712
Execution time: 15 msec
```

Instead of finding one solution of the problem we may try to find an optimal solution. For example, we may find a solution that maximizes the sum of all 4 variables in the array "vars". To do this it is enough to replace the line

```
   Solution solution = solver.findSolution();
```

with

```
   Solution solution =
```

```
solver.findOptimalSolution(Objective.MAXIMIZE,sum(vars));
```

The modified code will produce the results that may look like below:

```
Solution #8:
       X[4] Y[6] Z[10] R[9] sum[29]
```

Problem Definition Concepts

In the JSR-331 Problem Definition uses the following interfaces to represent a CSP with different constrained variables and constraints:

- **Problem**
- **Var**
- **VarBool**
- **VarReal**
- **VarSet**
- **Constraint**.

Below we describe major methods of the CP interfaces and provide examples of their use. The descriptions will include the column "Implementation Level" that states on which level (Common or CP solver) these methods should be implemented. The methods that are not normative are marked as "optional".

Interface "Problem"

The JSR-331 provides a generic interface Problem for any constraint satisfaction or optimization problem that allows a user to create and access major problem' objects. A problem serves as a factory for creation of constrained variables and constraints. Every variable and every constraint belongs to one and only one problem. For example, the code snippet

```
Problem p = new Problem("Test");
Var x = p.variable("X",1,10);
```

creates an instance "p" of the class `javax.constraints.impl.Problem` (defined by a particular JSR-331 implementation) and then the problem `p` creates a new constrained integer variable `x` with the domain [1,10] known under the name "X". Here the domain [1,10] is a set of integers from 1 to 10 without omissions. The variable `x` is automatically added to the problem.

The JSR-331 uses the Problem interface as a factory to standardize the signatures of the main methods that allow an end user to create constrained variables and constraints.

Creating Variables

All factory methods for constrained variables start with the word "`variable`" and newly created variables are always added to the problem. It means that you always may find the added variable using the method like `p.getVar("X")` and this variable will be automatically added to the default decision variables and future solutions (if any).

An alternative way to create an integer constrained variable (without adding it to the problem) is to use Var constructors defined in the implementation class

javax.constraints.impl.Var. For example, the code

```
Var x = p.variable("X",1,10);
```

is equivalent to the following

```
Var x = new Var(p,"X",1,10);
p.add(x);
```

The list of the main Problem's methods for creation of constrained integer variables is presented in the section "Creating Constrained Variables".

Creating and Posting Constraints

All factory methods for creating and posting constraints start with the word "post". Here is the current list of the method names used to create and post constraints:

- constraintLinear or simply constraint
- constraintAllDiff
- constraintElement
- constraintCardinality
- constraintGlobalCardinality
- constraintIfThen
- constraintMax
- constraintMin

Here are examples for creating and posting constraints:

```
post(x,"<",y);   // the same as postLinear(x,"<",y);
post(x.plus(y),"=",z);
postAllDifferent(vars);
postElement(vars,indexVar, "=", 5);
postCardinality(vars,3, ">", 0);
```

Thus, for the most popular linear constraints the suffix "Linear" in the method name "postLinear" may be omitted.

The linear constraint

```
post(x.plus(y),"=",z);
```

may also be created and posted using a constructor for the class "Linear" as follows:

```
new Linear(x.plus(y),"=",z).post();
```

However, in this case the program imports should additionally include

```
import javax.constraints.impl.constraint.Linear;
```

The use of constructors instead of Problem's constraint-methods is justified when constraints should be created without posting like in the following code:

```
// red bin contains at most 1 wood component
Constraint c1 = new Linear(type,"=",red);
Constraint c2 = new Linear(counts[wood],"<=",1);
postIfThen(c1,c2);
```

All JSR-331 implementations must provide their own constructors for constrained variables and constraints with these signatures. The interface `Problem` also includes convenience methods called "`linear`" that allow a user to create linear constraints without posting them. For example, the above example can be also presented as follows:

```
Constraint c1 = linear(type,"=",red);
Constraint c2 = linear(counts[wood],"<=",1);
postIfThen(c1,c2);
```

The standard specifies only commonly used forms of variable and constraint constructors, while an implementation may use other forms too. However, a user who decides to use non-standard constructors should be aware that s/he potentially makes a commitment to a selected implementation.

[Note for Reviewers. The initial standard version covers only the most popular constraints and more constraint creation methods will be added as the standard evolves.]

The lists of the main Problem's methods for constraint creation are presented in the section "Defining Constraints".

Common Methods

The Problem interface also specifies general methods for logging, versioning, creating a solver, and additional convenience methods – see the JSR-331 javadoc. Here are some of such methods:

Methods of the interface Problem	Impl. Level
`public String getAPIVersion();` This method returns the current version of the JSR-331 API	Common
`public String getImplVersion();` This method returns the current version of the concrete JSR-331 implementation	CP solver
`public Solver getSolver()` This method returns an instance of a Solver associated with this problem and that will be used to solve the problem. If a Solver's instance is not defined yet, this method creates a new Solver (lazy instantiation) and associates it with the problem.	Common

public void **log**(String text) This method logs (displays) the "text" to the default log (as defined by a selected implementation).	CP solver
public void **log**(Var[] vars) This method logs (displays) all variables from the array vars" to the default log.	CP solver
public Var **min**(Var var1, Var var2) This method returns a new variable constrained to be the minimum of variables var1 and var2	Common
public Var **max**(Var var1, Var var2) This method returns a new variable constrained to be the maximum of variables var1 and var2	Common
public Var **min**(Var[] vars); This method creates a new Var constrained to be the minimum of all variables in the array "vars"	Common
public Var **max**(Var[] vars); This method creates a new Var constrained to be the maximum of all variables in the array "vars"	Common
public Var **sum**(Var[] vars); This method creates a new Var constrained to be the sum of all variables in the array "vars"	Common
public Var **scalProd**(int[] values, Var[] vars); This method creates a new Var constrained to be the scalar product of the array of values and the array of variables "vars"	Common
public Var **element**(int[] values, Var indexVar); This method creates a new constrained variable that is an element of the integer array "values" with an index defined by another constrained variable "indexVar"	Common
public Var **element**(Var[] vars, Var indexVar); This method creates a new constrained variable that is an element of the array of constrained variables "vars" with an index defined by another constrained variable "indexVar"	Common

[Notes for Reviewers. When the proper standard for the CP XML is defined, the Problem interface will be expanded with two more methods that will allow a user to store/load a problem instance to/from am XML format:

```
public void storeToXML(OutputStream os, String comment) throws Exception;
public void loadFromXML(InputStream os) throws Exception;
```

At this stage of standardization, concrete implementations may implement these optional

methods using its own preferred XML format.

Question: do you think these two methods should be added to the interface with a stub that throws a runtime error defined in the common implementation?]

Common Interface "ConstrainedVariable"

The interface "ConstrainedVariable" defines common methods for all types of constrained variables. Here is a summary of these methods:

Method of the interface ConstrainedVariable	Comment	Impl. Level
`public void setName(String name)`	Defines the name of this variable	Common
`public String getName()`	Returns the name of this variable	Common
`public void setImpl(Object impl)`	This method defines a concrete implementation of this variable provided by a specific CP solver	Common
`public Object getImpl()`	This method returns a concrete implementation of this variable provided by a specific CP solver	Common
`public void setObject (Object obj)`	This method is used to attach a Business Object to this variable	Common
`public Object getObject()`	This methods returns a Business Object associated with this variable	Common

The methods `setObject` and `getObject` provide an ability to associate any application objects with constrained variables. These objects may be effectively used by application developers to define custom constraints and variable/value selectors.

The method `setImpl` is used by an underlying JSR-331 implementation to associate an implementation object with an instance of a standard constrained variable. It is used internally by JSR-331 implementations but also provides a user an ability to switch to an implementation level by using the method `getImpl()`. While it violates a solver independence principle, in certain situations a user still may want to take an advantage of a selected CP solver by casting implementation objects to solver specific classes and using them directly with additional methods provided by this particular solver.

The standard interface defines the following sub-interfaces of the common interface "ConstrainedVariable":

- **Var** (integer)
- **VarBool** (boolean)
- **VarReal** (floating-point)
- **VarSet** (set).

Note. At this early stage of the JSR-331 development, the sections below will mainly

concentrate on constrained integer variables. However, other types of constrained variables will be described in the next releases of this document.

Constrained Integer Variables "Var"

Constrained integer variables are the most popular type of the constrained variable (the reason why the name of this type "Var" does not have an additional identifier like "VarInt"). Each variable of the type Var has a finite domain of integer values.

The JSR-331 includes a common implementation of the major Var methods in the class

```
javax.constraints.impl.constraint.VarI
```

Each implementation must create its own class

```
javax.constraints.impl.constraint.Var
```

which should be inherited from the class `VarI` and that should be used to define constrained integer variables on the CP solver level.

Creating Integer Variables

The standard interface Problem provides multiple methods for creating new variables of the type Var. For example, a user may write:

```
Var digitVar = problem.variable("A", 0, 9);
```

A new constrained integer variable with an initial domain [0;9] will be created and added to the problem under the name "A". Here is the list of the major methods from the interface Problem that deal with creation and accessing constrained integer variables:

Methods of the interface Problem	Impl. Level
`public Var variable(String name, int min, int max)` This method creates a new Var with the name "name" and a domain [min;max]. It also adds a newly created variable to the problem, so later on you may find this variable by name using the Problem's method `getVar(name)`. There is a similar method without "name".	CP solver
`public Var variable(String name, int[] domain)` This method creates a new Var with the name "name" and a given "domain" (an array of regular integers). It also adds a newly created variable to the problem. There is a similar method without "name".	Common or CP solver
`public Var[] variableArray(String name, int min, int max, int size)` This method creates an array of constrained integer variable with the name like "name[i]" and a domain [min;max]. The total number of variables in the array is equal to "size". It also adds this array to the problem, so later on a user may find this array by name using the Problem's method `getVarArray(name)`.	Common

public Var **variable**(String name, **int** min, **int** max, DomainType type) This method creates a new Var with the name "name" and a domain [min;max]. The domain type of this variable is defined by the parameter "type" – see below. It also adds a newly created variable to the problem.	
public void setDomainType(DomainType type) This method sets a domain type (DOMAIN_SMALL, DOMAIN_MIN_MAX, DOMAIN_SPARSE, or DOMAIN_OTHER) that will be used as the default for subsequent creation of variables using var(…) and varArray(…) methods	Common
public Var **add**(Var var) This method adds already created variable of the type Var to the problem making its available for the for the proper method getVar("name"). All added variables will also be included in the future solutions of the problem.	Common
public Var **getVar**(String name) This method returns a Var that previously was added to the problem under the name "name"	Common
public Var[] **getVars**() This method returns all variables of the type Var that were previously added to the problem	Common

Any CP solver is expected to provide its own implementation of the class `javax.constraints.impl.Var` that extends the common abstract class `javax.constraints.impl.VarI`. It is expected that any implementation will at least support the standard constructor

```
public Var(Problem problem, String name, int min, int max)
```

It means that a user may also create and add a variable this way:

```
Var digitVar = new Var(problem, "A", 0, 9);
problem.add(digitVar);
```

Domain Types

While the standard VAR interface does not impose a particular organization of the Var's domain, it specifies different domain types using the following `enum`:

```
public enum DomainType {
    DOMAIN_SMALL,
    DOMAIN_MIN_MAX,
    DOMAIN_SPARSE,
    DOMAIN_OTHER
}
```

This classification assumes the following domain types:

DOMAIN_SMALL used for relatively small domains

DOMAIN_MIN_MAX used for large domains that mainly keep track of minimal and maximal values inside domains

DOMAIN_SPARSE used for domains with a lot of missing values between minimal and maximal values

DOMAIN_OTHER used for domains that may have a special meaning in any particular implementation.

A user may specify a certain domain type when creating a variable as follows:

```
Var var = variable("A",0,9,DomainType.DOMAIN_SMALL);
```

The common default domain type is *DOMAIN_SMALL* but an implementation may use a different default. A user may redefine a default domain type by using the following Problem's method:

```
public void setDomainType(DomainType type);
```

For example, if a user writes

```
setDomainType(DomainType.DOMAIN_SPARSE);
```

then all variables created after (!) this statement by default will have the domain type *DOMAIN_SPARSE*. After creating a few "sparse" variables, a user may switch to different domain type.

A user may also create constrained integers variables by listing all possible domain values like in this example:

```
int[] domain = new int[] {1,2,4,7,9};
Var var = variable("A", domain);
```

To create an array of 100 constrained integers variables with the domain [0;10], a user may write:

```
Var[] vars = variableArray("A", 0, 10, 100);
```

Note. Any JSR-331 implementation may provide other Var constructors that may take advantage of its specific features. At the same time a user should be warned that the use of CP solver specific constructors renders the application code dependent on that particular implementation. As more Var constructors become commonly acceptable for different implementations, they will be added to the standard Problem interface.

Manipulating Integer Variables

The Var interface provides the following methods that allow a user to evaluate the state of constrained integer variables:

- `int getDomainSize()` returns the current number of elements in the domain
- `DomainType getDomainType()` returns the domain type
- `boolean isBound()` returns true if the variable is already instantiated with a single value (domain's size is 1)

- `int getValue()` returns a value with which the variable was instantiated. If this variable is not bound, this method throws a runtime exception
- `int getMin()` returns the minimal value from the current domain
- `int getMax()` returns the maximal value from the current domain.

The JSR-331 does not allow a user to modify variables directly, e.g. using setters like "setMin" or "setMax" – they are simply not defined. Instead a user may only post the proper linear constraints:

`post(var,">=",min);` - to set the minimal value for the current domain
`post(var,"<=",max);` - to set the maximal value for the current domain
`post(var,"=",value);` - to instantiate the variable "var" with the "value"
`post(var,"!=",value);` - to remove a value from the variable domain.

Arithmetic Operations with Integer Variables

If a user wants to impose the constraint "x + y < 10", s/he can do it by posting this linear constraint

```
post(x.plus(y), "<", 10);
```

Here is the list of the arithmetic operations defined by the interface Var that create new constrained variables:

Methods of the interface Var	Impl. Level

`public Var plus(int value); // this + value`
This method creates a new Var constrained to be the sum of this variable and the given "value"
	CP solver

`public Var plus(Var var); // this + var`
This method creates a new Var constrained to be the sum of this variable and the given variable "var"
	CP solver

`public Var minus(int value); // this - value`
This method creates a new Var constrained to be the difference between this variable and the given "value"
	Common

`public Var minus(Var var); // this - var`
This method creates a new Var constrained to be the difference between this variable and the given variable "var"
	Common

`public Var multiply(int value); // this * value`
This method creates a new Var constrained to be the product of this variable and the given "value"
	CP solver

`public Var multiply(Var var); // this * var`
This method creates a new Var constrained to be the product of this variable and the given variable "var"
	CP solver

`public Var divide(int value);  // this / value`
This method creates a new Var constrained to be the quotient of this variable and the given "value". It throws a Runtime Exception if value = 0
	Common or CP solver

`public Var divide(Var var) throws Exception; // this / var`
This method creates a new Var constrained to be quotient of this variable and the given variable "var"
	Common or CP solver

`public Var mod(int value);  // this % value`
This method creates a new Var constrained to be the remainder after performing integer division of this variable by the given "value". It throws a Runtime Exception if value = 0
	Common

`public Var sqr();  // this * this`
This method creates a new Var constrained to be the product of this variable and itself
	Common

`public Var power(int value);  // this ** value`
This optional method creates a new Var constrained to be this variable raised to the power of the given "value"
	Common or CP solver

`public Var abs();  // abs(this)`
This method creates a new Var constrained to be the absolute value of this variable
	CP solver
	optional

Note that all these methods only create new constrained variables but do not add them to the problem. If necessary, it should be done explicitly with the Problem's method `add(var)`.

An end user should be warned that while the above operations could be convenient to create arithmetic expressions and then post constraints on them, these operations may create internally a lot of intermediate variables and constraints. For example, a user may represent constraint `3x + 4y -7z > 10` as

```
Var exp = x.multiply(3).plus(y.multiply(4)).minus(z.multiple(7));
post(exp, ">", 10);
```

However, it may be more efficient to use this constraint instead:

```
int[] coef1 = { 3, 4, -7 };
Var[] vars = { x, y, z };
post(coef1,vars, ">", 10);
```

Note. The names of the above operations correspond to the default names used by such dynamic languages as **Groovy** to allow operator overloading. So, for example the above constraint in Groovy may simply look as follows:

```
post(x*3+y*4-z*7, ">", 10);
```

Constrained Boolean Variables

Boolean variables of the standard type VarBool may be considered as integer variables

with domain [0;1] where 0 stand for “false” and 1 stands for “true”.

More details will be provided in the next releases.

Constrained Real Variables

More details will be provided in the next releases.

Constrained Set Variables

More details will be provided in the next releases.

Defining Constraints

The JSR-331 specifies many major constraints that define relationships between constrained variables. These constraints are available through the generic Problem interface. Here are examples of predefined constraints.

1. A constraint `x < y` between two constrained variables may be expressed as

   ```
   post(x, "<", y);
   ```

2. To express the fact a sum of all variables from the array “vars” of the type Var[] should be less than 20, a user may write:

   ```
   post(vars, "<", 20);
   ```

3. To express the fact that four variables x, y, z, and t are subject to the constraint

   ```
   3*x + 4y - 5*z  + 2*t >  x*y
   ```

   a user may create and post the following constraint:

   ```
   Var xy = x.multiply(y); // non-linear
   int[] coefs = { 3, 4, -5, 2 };
   Var[] vars = { x, y, z, t };
   post(coefs, vars, ">", xy);
   ```

4. If a user has an array of constrained variables “vars” and wants to state that all variables inside this array are different, s/he may write:

   ```
   postAllDifferent(vars);
   ```

All above examples use Problem’s factory methods starting with the word “post” to create and post(!) constraints. Posting a constraint means that this constraint will control the domain of all involved variables. Every time when constrained variables are modified the posted constraints defined on these variables will try to remove inconsistent values from their domains. This process is known as constraint propagation. If some domains become empty constraints throw exceptions. If an exception happens during the search then a search strategy will catch such exceptions and will react according to its own logic (e.g. continue to explore alternatives).

Depending on implementation, constraints may throw (or not) runtime exceptions during posting. In this case a user may put all constraint postings into a try-catch block to catch

contradictory constraints – see below. However, constraint posting by itself does not guarantee that all conflicts will be caught and it may require a search to find a solution or prove that all posted constraints actually cannot be satisfied.

5. To express the fact that three variables x, y, and z are subject to this constraint

   if (x > y) then z <= 5

   a user may write:

```
Constraint c1 = linear(x, ">", y);
Constraint c2 = linear(z, "<=", 5);
postIfThen(c1,c2);
```

Please note that contrary to `Constraint c1 = p.post(x, ">", y)` method "`linear`" only creates a constraint but does not post it. Instead of Problem's method "`linear`" a user also may equally use a constructor for a Linear constraint:

```
Constraint c1 = new Linear(x, ">", y);
```

In this case the program imports should additionally include

```
import javax.constraints.impl.constraint.Linear;
```

Common Interface "Constraint"

The interface "Constraint" defines common methods for all types of constraints. Here is a summary of these methods:

Method of the interface Constraint	Comment	Impl. Level
`void setName(String name)`	Defines the name of this constraint	Common
`String getName()`	Returns the name of this constraint	Common
`void setImpl(Object impl)`	This method defines a concrete implementation of this constraint provided by a specific CP solver	Common
`Object getImpl()`	This method returns a concrete implementation of this constraint provided by a specific CP solver	Common
`void setObject (Object obj)`	This method is used to attach any business object to this constraint	Common
`Object getObject()`	This methods returns a business object associated with this constraint	Common
`void post()`	This method is used to post the constraint. If the posting was unsuccessful, this method throws a runtime exception.	CP solver
`void post(ConsistencyLevel`	This method is used to post the constraint and also specifies a consistency level that controls the	CP solver

`consistencyLevel)`	propagation strength of this constraint (see below). If the posting was unsuccessful, this method throws a runtime exception.	optional
`Constraint` **`and`**`(Constraint c)`	This method creates a new constraint "and" that is satisfied only when "this" constraint and the parameter-constraint "c" are both satisfied	CP solver
`Constraint` **`or`**`(Constraint c)`	This method creates a new constraint "and" that is satisfied only when at least one of two constraints "this" or the parameter-constraint "c" is satisfied	CP solver
`Constraint` **`implies`**`(Constraint c)`	This method creates a new constraint that states: if this constraint is satisfied then parameter-constraint "c" should also be satisfied	CP solver
`Constraint` **`negation`**`()`	This method creates a new constraint that is satisfied if and only if this constraint is not satisfied	Common or CP solver
`VarBool` **`asBool`**`()`	This optional method returns a new constrained boolean variable that is equal 1 (true) if the constraint is satisfied and equals 0 (false) if it is violated.	CP solver optional

The methods `setObject` and `getObject` provide an ability to associate and use any application objects with a constraint..

The method `setImpl` is used by an underlying JSR-331 implementation to associate an implementation object with an instance of a standard constraint. It is used internally by JSR-331 implementations but also gives a user an ability to switch to an implementation level by using the method `getImpl()`. While it violates a solver independence principle, in certain situations a user still may want to take an advantage of a selected CP solver by casting implementation objects to solver specific constraint classes and using them directly with additional methods provided by this particular solver.

A user may create new constraints as combinations of the predefined constraints using the logical operations `and`, `or`, `implies`, and `negation`. While constraints can be either satisfied or not they may be considered as constrained Boolean variables and the method `asBool` allows a user to treat it as such. In particular, constraints as boolean variables be used may be used to define relative measures for different constraint violations (if any) and try to minimize the total violations – see an example below.

Note. Not all constraints must have implementations for the method `asBool()`. If a user will try to access a non-existing asBool method a runtime exception will be thrown. This behavior is defined in the common JSR-331 implementation and could be overridden by a concrete implementation.

Common Implementation "Constraint"

The JSR-331 common implementation provides the class

`javax.constraints.impl.constraint.ConstraintI` that already contains default implementations of some of the above methods. Each JSR-331 implementation should create its own class `javax.constraints.impl.constraint.Constraint` inherited from this class and that should be considered as a base class for all other constraints defined on the CP solver level.

There are several generic methods for creating and accessing constraints defined by the standardized Problem interface:

Methods of the interface Problem	Impl. Level
`Constraint` **`add`**`(Constraint constraint)` This method adds already created constraint to the problem making its available for the for the proper method getConstraint("name").	Common
`Constraint` **`getConstraint`**`(String name)` This method returns a Constraint that previously was added to the problem under the name "name"	Common
`Constraint[]` **`getConstraints`**`()` This method returns all constraints that were previously added to the problem	Common
`Constraint` **`post`**`(String name,` **`String symbolicExpression`**`)` This optional method creates a new constraint based on the "symbolicExpression" such as "x*y - z < 3*r". It is assumed that all variables in the expression were previously created under names used within this expression. The method adds this constraint to the problem and returns the newly added Constraint. This method throws a RuntimeException if there is either an error inside the "symbolicExpression" or there is no implementation for this method.	Common or CP solver optional

The Problem methods that create concrete constraints such as "Linear", "Element", "AllDifferent", "Cardinality", and "GlobalCardinality" are described below.

Posting Constraints

A constraint has no effect until it is posted. The constraint posting is implementation specific but usually it executes the following actions:

1) Initial constraint propagation (if any as defined by the common or solver specific implementation);

2) Associating constraints (or their propagators, listeners, observers – different implementations use different terms) with the involved constrained variables and

events. When such events occur the proper propagators will be woke up and executed to remove inconsistent values from the variable domains.

The actual posting logic depends on an underlying CP solver. If the posting was unsuccessful, the method `post` throws a runtime exception. So, it should be a regular practice to put constraint posting into a try-catch block, e.g.:

```
try {
      post(x,"<",y);              // X < Y
      post(x.plus(y),"=",z);   // X + Y = Z
      postAllDifferent(vars);
      int[] coef1 = { 3, 4, -7, 2 };
      post(coef1,vars,">",0); // 3x + 4y -7z + 2r > 0

} catch (Exception e) {
      log("Error posting constraint: " + e);
      System.exit(-1);
}
```

The standard allows a user to control the propagation strength of different constraints using an additional posting parameter in this method:

```
void post(ConsistencyLevel consistencyLevel)
```

Here the `ConsistencyLevel` is defined as the following standard enum:

```
public enum ConsistencyLevel {

      BOUND,      // bound consistency
      DOMAIN,     // domain consistency
      VALUE,      // value consistency
      OTHER       // implementation-specific consistency
}
```

The JSR-331 does not enforce any particular consistency level leaving this decision to implementers of different constraints. Note, that the common implementation simply ignores the consistency level, resulting in this method being equivalent to the regular `post()`.

Example of a Problem with Various Constraints

Usually application developers incorporate constrained variables in their own business objects and post constraints between them to express business relationships between yet unknown entities. Let's consider a popular problem: given a supply of different components and bins of given types, determine all assignments of components to bins satisfying specified assignment constraints subject to an optimization criterion. Here is a fragment of the business object Bin (a constructor only - the complete implementation is included in the standard package *org.jcp.jsr331.samples*):

```
static final int red = 0, blue = 1, green = 2;
static final int glass = 0, plastic = 1, steel = 2, wood = 3, copper = 4;

class Bin {
  public int id;
  public Var type;
```

```
  public Var capacity;
  public Var[] counts; // per component

  public Bin(Problem p, int binId) {
    id = binId;
    type = p.variable("Bin" + id + "Type", 0, binTypes.length - 1);

    p.log("Capacity constraints");
    int capacityMax = 0;
    for (int i = 0; i < binCapacities.length; i++) {
          if (binCapacities[i] > capacityMax)
                capacityMax = binCapacities[i];
    }
    capacity = p.variable("capacity", 0, capacityMax);
    p.postElement(binCapacities,type, "=", capacity);

    counts = new Var[components.length];
    for (int i = 0; i < components.length; i++)
          counts[i] = p.variable(countName(i), 0, capacityMax);
    // Sum of counts <= capacity
    p.post(counts, "<=", capacity);

    p.log("Containment constraints");
    Constraint c1, c2, c3;
    // red contains at most 1 of wood
    c1 = p.linear(type,"=",red);
    c2 = p.linear(counts[wood], "<=",1);
    c1.implies(c2).post();


    // green contains at most 2 of wood
    c1 = p.linear(type,"=",green);
    c2 = p.linear(counts[wood], "<=",2);
    c1.implies(c2).post();
    // red can contain glass, wood, copper
    c1 = p.linear(type,"=",red);
    c2 = p.linear(counts[plastic],"=",0);
    c3 = p.linear(counts[steel],"=",0);
    c1.implies(c2.and(c3)).post();
    // blue can contain glass, steel, copper
    c1 = p.linear(type,"=",blue);
    c2 = p.linear(counts[plastic],"=",0);
    c3 = p.linear(counts[wood],"=",0);
    c1.implies(c2.and(c3)).post();
    // green can contain plastic, wood, copper
    c1 = p.linear(type,"=",green);
    c2 = p.linear(counts[glass],"=",0);
    c3 = p.linear(counts[steel],"=",0);
    c1.implies(c2.and(c3)).post();
    // wood requires plastic
    c1 = p.linear(counts[wood],"=",0);
    c2 = p.linear(counts[plastic],"=",0);
    c1.implies(c2).post();
    // glass exclusive copper
    c1 = p.linear(counts[glass],"=",0);
    c2 = p.linear(counts[copper],"=",0);
    c1.or(c2).post();
    // copper exclusive plastic
    c1 = p.linear(counts[copper],"=",0);
    c2 = p.linear(counts[plastic],"=",0);
    c1.or(c2).post();
}
```

Linear Constraints

All constraints that deal with a comparison of constrained expressions use the standardized comparison operators expressed as strings:

```
"<"   // Less Than
"<="  // Less than or Equal to
"="   // Equal to
">="  // Greater than or Equal to
">"   // Greater Than
"!="  // Not Equal
```

Here is the list of **linear** constraints limited to constrained integer variables:

Methods of the interface Problem	Impl. Level
`Constraint` **`linear`**`(Var, String oper, int value)` This method creates and returns a new constraint such as "var < = value". For example, if "oper" is `"<="` it means that variable "var" must be less or equal to the "value".	CP solver
`Constraint` **`linear`**`(Var var1, String oper, Var var2)` This method creates and returns a new constraint such as "var1 < var2". For example, if "oper" is `"<"` it means that the variable "var1" must be less than the variable "var2".	CP solver
`Constraint` **`linear`**`(Var[] vars, String oper, int value)` This method creates and returns a new linear constraint such as "sum(vars) < = value". For example, if "oper" is `"<="` it means that a sum of all of the variables from the array "vars" must be less or equal to the "value".	CP solver
`Constraint` **`linear`**`(Var[] vars, String oper, Var var)` This method creates and returns a new linear constraint such as "sum(vars) < var". For example, if "oper" is `"<"` it means that a sum of all of the variables from the array "vars" must be less than the variable "var".	CP solver
`Constraint` **`linear`**`(int[] values, Var[] vars, String oper, int value)` This method creates and returns a new linear constraint such as "values*vars < value". For example, if "oper" is `"<"` it means that a scalar product of all "values" and all variables "vars" must be less than the "value". The arrays "values" and "vars" must have the same size otherwise a runtime exception will be thrown.	CP solver

`Constraint` **`linear`**`(int[] values, Var[] vars, String oper, Var var)` This method creates and returns a new linear constraint such as "values*vars < var". For example, if "oper" is "<" it means that a scalar product of all "values" and all variables "vars" must be less than the variable "var". The arrays "values" and "vars" must have the same size otherwise a runtime exception will be thrown.	CP solver

Instead of the methods with name "`linear`" a user may use the method "`constraint`" with the same parameters. In this case a constraint not only will be created but also posted.

When a user post the constraint "sum(vars) < 20" in this way:

```
post(vars, "<", 20);
```

there is no assumption that an intermediate variable for the "sum(vars)" will be created (it depends on a concrete constraint implementation). If a user actually needs this sum-variable, s/he may write a code similar to this one:

```
Var sumVar = variable("sum",min,max);
linear(vars, "=", sumVar).post();
post(sumVar, "<", 20);
```

or even easier:

```
post(sum(vars), "<", 20);
```

All Different Constraint

The JSR-331 interface Problem defines a simple way to create and post the most popular constraint commonly known as "allDifferent":

```
public Constraint postAllDifferent(Var[] vars);
```

There is also a more compact synonym:

```
public Constraint postAllDiff(Var[] vars);
```

These methods create, post, and return a new constraint stating that all constrained integer variables of the array "vars" must take different values from each other. There are similar methods for other types of variables. Another way to create and post "AllDiff" constraint is to use directly a constructor:

```
Constraint allDiff = new AllDifferent(Var[] vars);
allDiff.post();
```

In this case imports should additionally include

```
import javax.constraints.impl.constraint.AllDifferent;
```

Note. The latest example also allows posting with different consistency levels.

Element Constraints

The Problem interface also specifies convenience methods for creating constraints that

deal with elements of the arrays of constrained variables. If a constrained integer variable “indexVar” serves as an index within an array “values”, then the result of the operation “values[indexVar]” will be another constrained variable. While Java does not allow us to overload the operator “[ ]” the standard interface uses the Problem methods to create element constraints. Here is the list of such methods limited to integer variables (for now):

Methods of the interface Problem	Impl. Level
`Constraint` **`postElement`**`(int[] values, Var indexVar, String oper, int value)` This method creates, posts, and returns a new linear constraint such as “values[indexVar] < value”. Here "values[indexVar]" denotes a constrained integer variable whose domain consists of integer values[i] where i is within the domain of the "indexVar". For example, if “oper” is "<" it means that a variable “values[indexVar]” must be less than the “value”.	CP solver
`Constraint` **`postElement`**`(int[] values, Var indexVar, String oper, Var var)` This method creates, posts, and returns a new linear constraint such as “values[indexVar] < value”. Here "values[indexVar]" denotes a constrained integer variable whose domain consists of integer values[i] where i is within the domain of the "indexVar". For example, if “oper” is "<" it means that a variable “values[indexVar]” must be less than the variable “var”.	CP solver
`Constraint` **`postElement`**`(Var[] vars, Var indexVar, String oper, int value)` This method creates, posts, and returns a new linear constraint such as “vars[indexVar] < value”. Here "vars[indexVar]" denotes a constrained integer variable whose domain consists of integer values from the domain of the vars[i] where i is within the domain of the "indexVar". For example, if “oper” is "<" it means that a variable “vars[indexVar]” must be less than the “value”.	CP solver
`Constraint` **`postElement`**`(Var[] vars, Var indexVar, String oper, Var var)` This method creates, posts, and returns a new linear constraint such as “vars[indexVar] < value”. Here "vars[indexVar]" denotes a constrained integer variable whose domain consists of integer values from the domain of the vars[i] where i is within the domain of the "indexVar". For example, if “oper” is "<" it means that a variable “vars[indexVar]” must be less than the variable “var”.	CP solver

All possible comparison operators have been described above.

These constraints do NOT assume a creation of intermediate variables for "values[indexVar]" – the fact that may allow more efficient implementations.

Cardinality Constraints

The Problem interface specifies convenience methods for creating constraints that deal with cardinalities of the arrays of constrained variables. These constraints count how often certain values are taken by an array of constrained variables. The "cardinality variable" is a constrained variable that is equal to the number of those elements in the array "vars" that are bound to the value "cardValue". Here is the list of such methods limited to integer variables (for now):

Methods of the interface Problem	Impl. Level
`Constraint` **`postCardinality`**`(Var[] vars, int cardValue, String oper, int value)` This method creates, posts, and returns a new cardinality constraint such as "cardinality(vars,cardValue) < value". Here "cardinality(vars,cardValue)" denotes a constrained integer variable that is equal to the number of those elements in the array "vars" that are bound to the "cardValue". For example, if "oper" is *"<"* it means that the variable "cardinality(vars,cardValue)" must be less than the "value".	CP solver
`Constraint` **`postCardinality`**`(Var[] vars, int cardValue, String oper, Var var)` This method is similar to the one above but instead of "value" the "cardinality(vars,cardValue)" is being constrained by "var".	CP solver
`Constraint` **`postCardinality`**`(Var[] vars, Var cardVar, String oper, int value)` This method creates, posts, and returns a new cardinality constraint such as "cardinality(vars,cardVar) < value". Here "cardinality(vars,cardVar)" denotes a constrained integer variable that is equal to the number of those elements in the array "vars" that are equal to "cardVar". For example, if "oper" is *"<"* it means that the variable "cardinality(vars,cardValue)" must be less than the "value".	CP solver
`Constraint` **`postCardinality`**`(Var[] vars, Var cardVar, String oper, Var var)` This method is similar to the one above but instead of "value" the "cardinality(vars,cardVar)" is being constrained by "var".	CP solver

All possible comparison operators have been described above.

These constraints do NOT assume a creation of intermediate "cardinality" variables – the fact that may allow more efficient implementations.

Global Cardinality Constraints

The Problem interface also specifies convenience methods for creating global cardinality constraints (known as "gcc") that represent not one but multiple cardinalities at the same time. Here is the list of such methods limited to integer variables (for now):

Methods of the interface Problem	Impl. Level
`Constraint` **`postGlobalCardinality`**`(Var[] vars, int[] values, Var[] cardinalityVars)` This method creates and posts a new constraint that states: "For each index `i` the number of times the value `values[i]` occurs in the array `vars` is exactly `cardinalityVars[i]`" The arrays `cardinalityVars` and `values` should have the same size – otherwise a RuntimeException will be thrown. A newly created constraint is posted.	Common or CP solver
`Constraint` **`postGlobalCardinality`** `(Var[] vars, int[] values, int[] cardMin, int[] cardMax)` This method creates and posts a new constraint that states: "For each index `i` the number of times the value `values[i]` occurs in the array `vars` should be between `cardMin[i]` and `cardMax[i]` (inclusive)" The arrays `values`, `cardMin` and `cardMax` should have the same size – otherwise a RuntimeException will be thrown. A newly created constraint is posted.	Common or CP solver

The common JSR-331 implementation provides the default implementations of both these constraints using simple decompositions. Concrete implementation may (or may not) provide their own implementation class `GlobalCardinality` that supports both variants of this popular constraint with different consistency levels.

Min/Max Constraints

The Problem interface also specifies convenience methods for creating and posting constraints for constrained variables that are equal to a minimum and a maximum of other variables.

Methods of the interface Problem	Impl. Level
`Constraint` **`postMin`**`(Var[] vars,String oper,int value)` This method creates and posts a new constraint that states: "The minimal variable in the array `vars` should be less that `value`" if the `oper` is "<". Replace the word "less" for the proper words for all other comparison operators. A newly created constraint is posted.	Common
`Constraint` **`postMin`**`(Var[] vars, String oper, Var var)` This method creates and posts a new constraint that states: "The minimal variable in the array `vars` should be less that `var`" if the `oper` is "<". Replace the word "less" for the proper words for all other comparison operators. A newly created constraint is posted.	Common

There are similar constraints **`postMax`** defined for maximal variables in the array `vars`.

More Constraints

Any JSR-331 implementation is expected to provide its own implementations of major constraints specified in the standard interface Problem. The JSR-331 TCK (Technology Compatibility Kit) will check that a compliant implementation supports at least the basic forms of the constraints described above.

At the same time as the standard evolves, JSR-331 implementations may provide other constructors for already defined constraints and for other constraints they have implemented. The only requirement is that constraints not included in the standard should still implement the interface `javax.constraints.Constraint`. This approach will allow a user to take advantage of the implementation-specific features. At the same time a user should be warned that the use of implementation specific constructors renders the application code dependent on that particular implementation.

The common JSR-331 implementation `javax.constraints.impl.constraint` already provides several additional constraints that do not depend on a particular CP solver. Among them:

- `ConstraintTrue`: always successful
- `ConstraintFalse`: always fails when posted
- `ConstraintTraceVar`: used by the common Solver to implement methods trace(..)
- `ConstraintMax`: provides a constraint for a maximum of an array of constrained variables
- `ConstraintMin`: provides a constraint for a minimum of an array of constrained variables
- `ConstraintNotAllEqual`: provides a constraint that states that not all elements inside an array of constrained variables are the same or all equal to the values from a given array of integers.

More similar constraints will be added to the common JSR-331 implementation as the standard evolves.

[Notes for Reviewers. There are more than 300 constraints described at the Global Constraint Catalog. As more constraints become commonly acceptable for different implementations, they will be moved to the common JSR-331 level. The constraints "regular", "diffn", and "cumulative" are among the next to be considered. Which constraints and in which form will you recommend to add to the standard first?]

User-Defined Constraints

A user can define problem-specific constraints by combining the existing constraints using Constraint logical operations "`and`", "`or`", "`negation`", and "`implies`" defined in the interface Problem.

JSR-331 users also may create a subclass of the common predefined class `javax.constraints.impl.constraint`.ConstraintI to define their own constraints. For example, here is an example of the constraint `ConstraintNotAllEqual` that actually defines two constraints:

1) not all elements inside an array of constrained variables are the same

2) not all elements inside an array of constrained variables are equal to the values from a given array of integers.

```
//============================================
// J A V A  C O M M U N I T Y  P R O C E S S
//
// J S R  3 3 1
//
// Common Implementation
//
//============================================
package javax.constraints.impl.constraint;

import javax.constraints.Constraint;
import javax.constraints.Oper;
import javax.constraints.Var;
import javax.constraints.VarBool;
import javax.constraints.impl.ConstraintI;

public class ConstraintNotAllEqual extends ConstraintI {

   Constraint constraint;

   public ConstraintNotAllEqual(Var[] vars) {
        super(vars[0].getProblem());
        Problem p = getProblem();
        int n = vars.length-1;
        VarBool[] equalities = new VarBool[n];
        for (int i = 0; i < n; i++) {
           equalities[i] = p.linear(vars[i],"=",vars[i+1]).asBool();
        }
        constraint = p.linear(equalities, "<", n);
   }

   public ConstraintNotAllEqual(Var[] vars, int[] values) {
         super(vars[0].getProblem());
         Problem p = getProblem();
         if (values.length != vars.length)
               throw new RuntimeException(
             "ConstraintNotAllEqual requires arrays of the same length");
         int n = vars.length;
         VarBool[] equalities = new VarBool[n];
         for (int i = 0; i < n; i++)
           equalities[i]=p.linear(vars[i],"=",values[i+1]).asBool();
         constraint = p.linear(equalities, "<", n);
   }

   public void post() {
```

```
            constraint.post();
        }
    }
```

[Question for Reviewers. Should the standard give a user an access to more advanced concepts such as Propagators and Propagation Events to create new custom constraints as the JSR-331 evolves?]

PROBLEM RESOLUTION CONCEPTS

To represent the Problem Resolution part of any CSP, the JSR-331 uses the interface "**Solver**". The solver allows a user to solve the problem by finding feasible or optimal **Solution**s. Here is an example of a simple problem resolution:

```
problem.log("=== Find One solution:");
Solver solver = problem.getSolver();
Solution solution = solver.findSolution();
if (solution != null)
  solution.log();
else
  problem.log("No Solutions");
```

In this simple case, the default solver (defined as an instance of the class `javax.constraints.Solver`) is trying to find one solution using the default search strategy that enumerates all variables previously added to the problem. The JSR-331 explicitly defines the interface "`SearchStrategy`" that can be adjusted by a user and used by the solver to find solutions of the problem.

Interface "Solver"

The JSR-331 provides interface "`java.constraints.Solver`" (and its common implementation "`java.constraints.impl.search.SolverI`") that specifies different problem resolution concepts and methods. It is possible to create multiple solvers for the same problem. These solvers may produce different solutions pursuing different objectives. During the execution of Solver's methods the state of the Problem can be changed. The interface Solver provides the following enum to control a problem' state after the solver execution:

```
public enum ProblemState {
      RESTORE,
      DO_NOT_RESTORE
}
```

Another enum "Objective" provided by the interface Solver is

```
public enum Objective {
      MINIMIZE,
      MAXIMIZE
}
```

It allows a user to specify the optimization objective within the method "findOptimalSolution".

Below is the list of the major methods from the interface Solver:

Method of the interface Solver	Impl. Level
public Solution **findSolution**(); This method attempts to find a solution of the problem, for which the solver was defined. It uses the default search strategy or the strategy defined by the latest method setSearchStrategy(). It returns the found solution (if any) or null. If a solution is found, all decision variables will remain instantiated with the solution values after the execution of this method. If a solution was not found, the problem state will be restored.	Common
public Solution **findSolution**(ProblemState restoreOrNot); This method attempts to find a feasible solution of the problem, for which the solver was defined. It uses the default search strategy or the strategy defined by the latest method setSearchStrategy (). It returns the found solution (if any) or null. If a solution is not found, the problem state is restored. If a solution is found, the problem state will be restored only if the parameter "restoreOrNot" is *RESTORE*. If the parameter "restoreOrNot" is *DO_NOT_RESTORE*, after a solution is found all decision variables will be instantiated with the solution values.	CP solver
public Solution **findOptimalSolution**(Objective objective, Var objectiveVar); This method attempts to find the solution that minimizes/maximizes the objective variable "objectiveVar". The first parameter could have one of two values: Objective.MINIMIZE or Objective.MAXIMIZE. To find solutions this method uses the default search strategy or the strategy defined by the latest method setSearchStrategy(). The optimization process can be controlled by: - OptimizationTolerance that is a difference between solution objectives during two consecutive process iterations - see the method setOptimizationTolerance() - MaxNumberOfSolutions that is the total number of considered solutions - may be limited by the method setMaxNumberOfSolutions() - TimeLimit that is the total number of milliseconds allocated for the entire optimization process as it can be set by the method setTimeLimit(). The problem state after the execution of this method is always restored. The produced optimal solution (if any) will contain found values for all variables that were added to the problem (including the objectiveVar).	Common or CP solver
public Solution **findOptimalSolution**(Var objectiveVar); This method is an equivalent of findOptimalSolution(Objective.MINIMIZE,objectiveVar)	Common

public Solution[] **findAllSolutions**(); This method attempts to find all solutions for the Problem. It uses the default search strategy or the strategy defined by the latest method setSearchStrategy(). It returns an array of found solutions or null if there are no solutions. A user has to be careful not to overload the available memory because the number of found solutions could be huge. The process of finding all solutions can be also controlled by: - OptimizationTolerance that is a difference between solution objectives during two consecutive process iterations - see the method `setOptimizationTolerance()` - MaxNumberOfSolutions that is the total number of considered solutions - may be limited by the method `setMaxNumberOfSolutions()` - TimeLimit that is the total number of milliseconds allocated for the entire optimization process as it can be set by the method `setTimeLimit()`. The common implementation is based on the SolutionIterator (see below).	Common or CP solver
public SolutionIterator **solutionIterator**(); This method creates and returns a solution iterator that allows a user to find and navigate through multiple solutions (if any) using the current search strategy.	Common or CP solver
public void setSearchStrategy(SearchStrategy strategy); This method sets a search strategy defined as a parameter as a new default search strategy to be used by methods findSolution(), findOptimalSolution(..), findAllSolutions(..), and by solution iterators. At least one search strategy should be defined by every implementation as the default search strategy.	Common
public SearchStrategy **getSearchStrategy**(); This method returns the current search strategy that was set by an implementation as the default search strategy or by the latest call of the method `setSearchStrategy()`. A user may adjust the search strategy by changing its default decision variables, its variable selector, and/or its value selector. Search strategy are used by methods findSolution(), findOptimalSolution(..), findAllSolutions(..), and by solution iterators.	Common
public SearchStrategy **newSearchStrategy**(); This method returns a new instance of the search strategy that is set by an implementation as the default search strategy. A user may adjust this search strategy by changing its default decision variables, its variable selector, and/or its value selector. This new strategy may be added to the strategy execution list using the Solver's method "addStrategy"	CP solver
public void **addSearchStrategy**(SearchStrategy strategy); This method adds the strategy to the end of the strategy execution list.	CP solver

public void setMaxNumberOfSolutions(int number); This method sets a limit for a number of solutions that can be found by the method "findAllSolutions" or can be considered during execution of the method "findOptimalSolution". The default value is -1 that means there are no limits for a number of considered solutions.	Common
public int getMaxNumberOfSolutions(); This method returns a number that was set by the method setMaxNumberOfSolutions(...)	Common
public void setOptimizationTolerance(int tolerance); This method specifies a tolerance for the method "findOptimalSolution". If the difference between newly found solution and a previous one is less or equal to the "tolerance" then the last solution is considered to be the optimal one. By default, the optimization tolerance is 0.	Common
public int getOptimizationTolerance(); This method returns a tolerance that was set by the method setOptimizationTolerance(...)	Common
public void setTimeLimit(int milliseconds); This method specifies a time limit in milliseconds for the total execution of different find-methods. By default, there is no time limit.	Common
public int getTimeLimit(); This method returns a time limit in milliseconds for the total execution of different find-methods. By default, it returns -1 that means there is no time limit.	Common
public void logStats(); This method logs the solver execution statistics such as a number of choice points, number of failures, used memory, etc. This method is expected to be specific for different implementations. By default only time information will be logged out.	Common or CP solver optional

The Solver interface also defines several other convenience methods such as tracing methods:

- trace(Var var)
- trace(Var[] vars)
- traceFailures(boolean yesno)
- traceExecution(boolean yesno).

[Question to Reviewers. Currently the standard limits only a maximal number of considered solutions and allows a user to set time limits. Should a user be able to set limits for a number of choice points, failures, and other search characteristics? Concrete suggestions?]

Example of Constraint Relaxation Problem

The following example demonstrates how to deal with real-world situations when some constraints should be relaxed to make the problem solvable. It also demonstrates how to find an optimal solution of the problem that in this case is a solution that minimizes the total constraint violation.

Consider a map coloring problem that involves choosing colors for the countries on a map in a such way that no two neighboring countries have the same colors. When there are not enough colors some of these constraints have to be violated based of their relative importance. Below is a solution of this problem as it is presented in the JSR-331 TCK.

```
package org.jcp.jsr331.samples;
import javax.constraints.Var;
import javax.constraints.Solution;
import javax.constraints.impl.Problem;

public class MapColoringWithViolations extends Problem {
   static final String[] colors = { "red", "green", "blue" };

   public MapColoringWithViolations() {
      try {
         // Variables
         int n = colors.length-1;
         Var Belgium     = variable("Belgium",0, n);
         Var Denmark     = variable("Denmark",0, n);
         Var France      = variable("France",0, n);
         Var Germany     = variable("Germany",0, n);
         Var Netherlands = variable("Netherland",0, n);
         Var Luxemburg   = variable("Luxemburg",0, n);
         Var[] vars =
            {Belgium,Denmark,France,Germany,Netherlands,Luxemburg};
         // Hard Constraints
         post(France, "!=",Belgium);
         post(France, "!=",Germany);
         post(Belgium,"!=",Netherlands);
         post(Belgium,"!=",Germany);
         post(Germany,"!=",Netherlands);
         post(Germany,"!=",Denmark);
         // Soft Constraints
         Var[] weightVars = {
            linear(France, "=",Luxemburg).asBool().multiply(257),
            linear(Luxemburg,"=",Germany).asBool().multiply(9043),
            linear(Luxemburg,"=",Belgium).asBool().multiply(568)
         };
         // Optimization objective
         Var weightedSum = sum(weightVars);
         weightedSum.setName("Total Constraint Violations");

         Solution solution =
            getSolver().findOptimalSolution(weightedSum);
         if (solution != null) {
               solution.log();
               for (int i = 0; i < vars.length; i++) {
                  String name = vars[i].getName();
```

```
                    log(name+" - "+colors[solution.getValue(name)]);
                }
            }
            else
                log("no solution found");

        } catch (Exception ex)
                ex.printStackTrace();
    }
}
```

This problem may produce the results that may look like below;

```
Solution #1:
Belgium[0] Denmark[0] France[1] Germany[2] Netherland[1]
Luxemburg[1] Total Constraint Violations[257]
Belgium - red
Denmark - red
France - green
Germany - blue
Netherland - green
Luxemburg - green
```

Interface "SearchStrategy"

The JSR-331 utilizes the concept "SearchStrategy" to allow a user to choose between different search algorithms provided by different implementations. Search strategies are used by those Solver's methods that find a solution, find all solutions, find an optimal solution, and by solution iterators. A search strategy should know all decision variables it will try to instantiate during the search, and may need external selectors for variables and values. At least one decision strategy should be provided by any implementation to serve as the default strategy created in the implementation specific Solver constructor. The common interface "SearchStrategy" defines the following methods:

Methods of the interface "SearchStrategy"	Impl. Level
public void setName(String name) **public String getName**() Define a setter and a getter for the name of this strategy	Common
public Solver **getSolver**() Returns a solver with which this strategy is associated.	Common
public void setType(SearchStrategyType type) Sets a type for this strategy. The specification currently defined two type: SearchStrategyType.*DEFAULT* and SearchStrategyType.*CUSTOM*	Common
public Var[] **getVars**(); This method returns an array of integer variables that are used by the strategy.	Common

`public void setVars(Var[] vars);` This method sets an array of integer variables that will be used by the strategy.	Common
`public void setVarSelector(VarSelector selector);` This method sets a variable selector that will be used by the strategy during the search.	Common
`public void setVarSelectorType(VarSelectorType type);` This method sets a variable selector of the standard type specified as a parameter.	Common
`public void setValueSelector(ValueSelector selector);` This method sets a value selector that will be used by the strategy during the search.	Common
`public void setValueSelectorType(ValueSelectorType type);` This method sets a value selector of the standard type specified as a parameter.	Common
`public VarReal[] getVarReals();` This method returns an array of real variables that are used by the strategy.	Common
`public void setVarReals(VarReal[] vars);` This method sets an array of real variables that will be used by the strategy.	Common
`public VarSet[] getVarSets();` This method returns an array of set variables that are used by the strategy.	Common
`public void setVarSets(VarSet[] vars);` This method sets an array of set variables that will be used by the strategy.	Common
`public void trace();` This method forces the strategy to trace itself during the execution.	CP solver

All implementation specific search strategies should be implemented as subclasses of the common base class "`javax.constraints.impl.search.SeachStrategyI`". The only requirement to all search strategies is the following: when they are invoked by an implementation specific Solver method `findSolution(ProblemState state)` they are expected to either produce a solution of the problem within the current time limit or to report that a solution cannot be found. How they do it and how the internal interaction between Solver method "findSolution" and its search strategy is organized remains a prerogative of a concrete JSR-331 implementation.

Strategy Execution List

The search strategy execution list allows a user to mix strategies for different types of decision variables and to control their execution order. For example, for scheduling and resource allocation problems a user may decide first to schedule all activities and then assign resources to already scheduled activities. But a user may also decide first to assign resources and only then to schedule activities based on resource availability.

The Solver method

```
SearchStrategy getSearchStrategy()
```

returns the first search strategy specific for this particular JSR-331 implementation. A user may specify decision variables and set different variable selectors and value selectors for the default strategy. The Solver method

```
SearchStrategy newSearchStrategy()
```

returns a new instance of the default search strategy specific for this particular JSR-331 implementation. A user may specify decision variables and selectors for this strategy and then add it to the end of the "search strategy execution list". The Solver executes all strategies from this list in the order they were added, and the execution succeeds only when all strategies from the list are successfully executed.

Let's assume that a user has two arrays of decision variables "types" and "counts" and wants Solver first instantiate all types and only then all counts (possibly using different selectors). Here is how it can be done:

```
Solver solver = problem.getSolver();
SearchStrategy typeStrategy = solver.getSearchStrategy();
typeStrategy.setVars(types);
SearchStrategy countStrategy = solver.newSearchStrategy();
countStrategy.setVars(counts);
countStrategy.setVarSelectorType(VarSelectorType.MIN_DOMAIN);
solver.addSearchStrategy(countStrategy);
solution = solver.findSolution();
```

There are several convenience methods that allow a user to add additional strategies to the execution list without explicitly creating new search strategies. The method "addSearchStrategy" also supports different combinations of parameters Var[], VarSelector, and ValueSelector. The above code may be written more compactly as:

```
Solver solver = problem.getSolver();
solver.getSearchStrategy().setVars(types);
solver.addSearchStrategy(counts, VarSelectorType.MIN_DOMAIN);
solution = solver.findSolution();
```

Adding Non-Search Strategies

The JSR-331 allows a user to add a non-search strategies to the search strategy execution list. The common JSR-331 implementation provides an example of such non-search strategy called `StrategyLogVariables`. A user may use this strategy to display a state of problem variables after different search iterations like in this example:

```
Solver solver = getSolver();
solver.addSearchStrategy(new StrategyLogVariables(solver));
Solution solution =
   solver.findOptimalSolution(Objective.MAXIMIZE, getVar("cost"));
if (solution != null)
   solution.log();
else
   log("No Solutions");
```

In this case every time when the default search strategy finds a solution the `StrategyLogVariables` will show the state of all problem variables until an optimal

solution will be found. Here is the implementation code from the file `StrategyLogVariables.java`:

```
package javax.constraints.impl.search;
import javax.constraints.Solver;
import javax.constraints.Var;

public class StrategyLogVariables extends SearchStrategyI {
   Var[] vars;

   public StrategyLogVariables(Var[] vars) {
         super(vars[0].getProblem().getSolver());
         this.vars = vars;
         setType(SearchStrategyType.CUSTOM);
   }

   public StrategyLogVariables(Solver solver) {
         super(solver);
         vars = getProblem().getVars();
         setType(SearchStrategyType.CUSTOM);
   }
   public boolean run() {
         getProblem().log("=== StrategyLogVariables:");
         getProblem().log(vars);
         return true;
   }
}
```

A user may write in a similar way his/her own non-search strategy for displaying or saving intermediate search results including application specific objects. It is important to define a strategy type as `SearchStrategyType.CUSTOM`.

Variable Selectors

The JSR-331 specifies a set of standard variable selectors that can be used by an end user to customize the standard search strategy. These variable selectors are defined by the standard interface "VariableSelector" using the following enum:

```
static public enum VarSelectorType {
   /**
    * selection of variables in order of definition
    */
   INPUT_ORDER,

   /**
    * smallest lower bound
    */
   MIN_VALUE,

   /**
    * largest upper bound
    */
   MAX_VALUE,

   /**
```

```
     * min size of domain, tie break undefined
     */
    MIN_DOMAIN,

    /**
     * min size of domain, smallest lower bound tie break
     */
    MIN_DOMAIN_MIN_VALUE,

    /**
     * min size of domain, random tie break
     */
    MIN_DOMAIN_RANDOM,

    /**
     * random selection of variables
     */
    RANDOM,

    /**
     * min size of domain as first criteria, tie break by degree
     * that is the number of attached constraints
     */
    MIN_DOMAIN_MAX_DEGREE,

    /**
     * min value of fraction of domain size and degree
     */
    MIN_DOMAIN_OVER_DEGREE,

    /**
     * min value of domain size over weighted degree
     */
    MIN_DOMAIN_OVER_WEIGHTED_DEGREE,

    /**
     * largest number of recorded failures in attached constraints
     */
    MAX_WEIGHTED_DEGREE,

    /**
     * largest impact, select variable which when assigned restricts
     * the domains of all other variables by the largest amount
     */
    MAX_IMPACT,

    /**
     * largest number of attached constraints
     */
    MAX_DEGREE,

    /**
     * largest difference between smallest
     * and second smallest value in domain
     */
    MAX_REGRET,
```

```
    /**
     * custom variable selector
     */
    CUSTOM
}
```

Not all these selectors have to be implemented by every JSR-331 implementation. Most of variable selectors have been already included in the common implementation in the package "javax.constraints.impl.search.selectors". However, the variable selectors *MIN_DOMAIN_OVER_WEIGHTED_DEGREE*, and *MAX_WEIGHTED_DEGREE* are optional and may be implemented by a particular implementation only.

To set a new variable selector such as *MIN_DOMAIN*, a user may write:

```
SearchStrategy strategy = solver.setSearchStrategy();
strategy.setVarSelectorType(VarSelectorType.MIN_DOMAIN);
```

A user can easily implement her own variable selector as a subclass of the standard class "javax.constraints.impl.search.selectors.VarSelectorI" by overloading only this abstract method:

```
/**
* Returns the index of the selected variable in the array
* of constrained variables passed to the selector as a
* constructor' parameter.
* If no variables to select, it returns -1;
*/
abstract public int select();
```

Such custom selector can take into consideration the business objects potentially attached to every constrained variable.

Similar selectors for other types of constrained variables will be added later on.

Value Selectors

The JSR-331 specifies a set of standard value selectors that can be used by an end user to customize the standard search strategy. These value selectors are defined by the standard interface "ValueSelector" using the following enum:

```
static public enum ValueSelectorType {
    /**
     * try values in increasing order one at a time
     * without removing failed values on backtracking
     */
    IN_DOMAIN,

    /**
     * try values in increasing order, remove value on backtracking
     */
    MIN,

    /**
```

```
     * try values in decreasing order, remove value on backtracking
     */
    MAX,

    /**
     * try to alternate minimal and maximal values
     */
    MIN_MAX_ALTERNATE,

    /**
     * try values in the middle of domain,
     * the closest to (min+max)/2
     */
    MIDDLE,

    /**
     * try the median values first,
     * e.g if domain has 5 values, try the third value first
     */
    MEDIAN,

    /**
     * try a random value
     */
    RANDOM,

    /**
     * try a value which causes the smallest domain reduction
     * in all other variables
     */
    MIN_IMPACT,

    /**
     * custom value selector
     */
    CUSTOM
}
```

Not all these selectors have to be implemented by every JSR-331 implementation. Most of the value selectors are already included in the common implementation in the package "`javax.constraints.impl.search.selectors`". However, the value selectors *`IN_DOMAIN`* and *`MIN_IMPACT`* are optional and may be implemented by a particular implementation only.

To set a new variable selector such as *`MEDIAN`*, a user may write:

```
SearchStrategy strategy = solver.getSearchStrategy();
strategy.setValueSelectorType(ValueSelectorType.MEDIAN);
```

A user can easily create her own value selector by implementing the standard interface "`javax.constraints.ValueSelector`" with only two methods:

```
/**
```

```
 * Returns a value from the domain of constrained variable "var"
 */
public int select(Var var);

/**
 * Returns a type of this value selector
 */
public ValueSelectorType getType() {
    return ValueSelectorType.CUSTOM;
}
```

Such custom selectors can take into consideration the business objects potentially attached to every constrained variable.

Similar selectors for other types of constrained variables will be added later on.

More Search Strategies

At this stage of the JSR-331 development, the only way for a user to utilize search strategies different from the default one is to become implementation dependent. For example, if an implementation provide a BoundBacktrackingSearchStrategy as a subclass of "`javax.constraints.impl.search.SeachStrategyI`", then to use this strategy a user may write:

```
SearchStrategy strategy =
              new BoundBacktrackingSearchStrategy(100); // steps
solver.setSearchStrategy(strategy);
```

[Notes for Reviewers. As the standard evolves and more implementations offer more common search strategies, they will be added to the standard Solver interface (it is similar to addition of more global constraints to the Problem interface). The following search strategies are considered as optional strategy-candidates to be added to the standard down on the road:

- ***RestartSearchStrategy**(RestartFunction): for every run, the RestartFunction method tells how many choices to explore. If no solution is found, the RestartFunction method is called again to find how many choices should be allowed in the next run.*

- ***BoundedBacktrackingSearchStrategy**(Steps): search for a limited number of steps, which is given as a parameter; if no solution is found the strategy fails.*

- ***LimitedDiscrepancySearchStrategy**(Disc): search all possible assignments which differ in exactly Disc choices from the heuristic choice; typically used repeatedly with increasing discrepancy for values 0,1,2 and sometimes 3*

- ***CreditBasedSearchStrategy**(InitialCredit,CreditFunction,Steps): at the top of the tree, explore nodes by distributing credit from parent to children (not always equal split between them) using the CreditFunction method. When the credit runs out, allow a bounded backtracking search of Steps failures before giving up on this*

branch and returning to the credit part

- ***DepthBoundedSearchStrategy****(Level,Steps): explore the top Level levels of the search tree completely, below allow a bounded backtracking search with Steps failures.*

This list is provided to initiate a constructive discussion among experts. Even when more strategies are added to the standard, not all of them will have to be supported by every implementation to be JSR-331 compliant.]

Interface "Solution"

The standard interface "Solution" specifies solutions can be generated by such Solver' methods as "findSolution", "findOptimalSolution", and by solution iterators. This interface is completely implemented on the common level in the class "`javax.constraints.impl.search.SolutionI`" but any JSR-331 implementation may extend it with its own subclass "`javax.constraints.impl.search.Solution`".

A solution instance contains copies of all decision variables that were used by a search strategy that created this solution. These copies are in the state, in which original variable would be left after the solution search is completed but before a possible state restoration. There are no requirements that all decision variables should be instantiated – it depends on the used search strategy.

Here are main Solution's methods:

Method of the interface Solution	Impl. Level
`public Var[] getVars();` This method returns an array of variables with the same names as all variables that were added to the problem. These variables keep a current state of the initial variables when the solution was found.	Common
`public Var getVar(String name);` This method returns the variable with the name "name" saved within this solution. It throws a runtime exception if the proper variable does not exist. It is a copy of the actual problem's variable with the name "name" but it is in the state in which original variable would be left after the solution search is completed before a possible state restoration.	Common
`public int getValue(String name);` This method returns the found value of the variable with the name "name" saved within this solution. It throws a runtime exception if the proper variable does not exist or was not instantiated during the solution search.	Common
`public boolean isBound();` This method returns true only if all solution variables are instantiated (bound).	Common

`public boolean isBound(String name);` This method returns true only if a solution' variable with the given name is bound.	Common
`public int getSolutionNumber();` This method returns a number associated with this solution. Solution numbers start with 1.	Common
`public void setSolutionNumber(int number);` This method sets a solution number. This method is to be used by a solution strategy that creates this solution.	Common
`public void log();` This method logs all Solution's variables to the Problem's log. These variables have shown in the state as there were when the solution was found (some variables could remain non-instantiated).	Common
`public Solver getSolver();` This method returns a solver which generated this solution.	Common

There are similar methods for other types of variables.

Solution Iterator

The standard interface `SolutionIterator` allows a user to find and iterate through multiple solutions and execute different application specific actions with each found solution. The intended use of a solution iterator is presented by the following code:

```
SolutionIterator iter = solver.solutionIterator();
while(iter.hasNext()) {
   Solution solution = iter.next();
   ...
}
```

For example, a solution iterator may be uses to provide a very simple implementation of the Solver's method "`findAllSolutions`":

```
public Solution[] findAllSolutions() {
       SolutionIterator iter = solutionIterator();
       ArrayList<Solution> solutions = new ArrayList<Solution>();
       while(iter.hasNext()) {
              Solution solution = iter.next();
              solutions.add(solution);
       }
       Solution[] array = new Solution[solutions.size()];
       for (int i = 0; i < array.length; i++) {
              array[i] = solutions.get(i);
       }
       return array;
}
```

The common implementation also takes into consideration the current limits for a

maximal number of solutions and for the total available time. This code provides an example of how a user may navigate through different solutions. A user may add its own code to decide which solutions to save and when to stop the search.

In a similar way, we a user may implement its own search for an optimal solution:

```
public Solution findOptimalSolution(Var objectiveVar) {

    SolutionIterator iter = solutionIterator();
    int bestValue = Integer.MAX_VALUE;
    Solution solution = null;
    while(iter.hasNext()) {
           solution = iter.next();
           try {
                  int newValue = solution.getValue(objectiveVar.getName());
                  if (bestValue > newValue)
                         bestValue = newValue;
                  getProblem().post(obj,"<",newValue); // may fail
           } catch (Exception e) {
                  break;
           }
    }
    objectiveVar.setName(oldName);
    return solution;
}
```

The common implementation of this method in the package "`javax.constraints.impl.search.SolverI`" also takes into consideration the current limits for a maximal number of solutions and for the total available time.

The `Objective.MAXIMIZE` can be replaced by `Objective.MINIMIZE` for the objectiveVar that is opposite to the original objective.

These implementations are given only as examples for end users who may organize their own solution iteration cycles. For example, a user may decide to find 3 best solutions within 10 seconds. It becomes a matter of setting the proper filters inside the above main loop right after `iter.next()`. A user may also utilize business objects associated with decision variables to compare different solutions.

Note that the described implementations can be used with any search strategy.

Below is in a very simplified (and inefficient but working) implementation of the interface `SolutionIterator`:

```
public class SolutionIteratorI implements SolutionIterator {

    Solver solver;
    Solution solution;
    int solutionNumber;
    boolean noSolutions;

    public SolutionIteratorI(Solver solver) {
           this.solver = solver;
           solution = null;
           noSolutions = false;
           solutionNumber = 0;
    }

    public boolean hasNext() {
```

```
            if (noSolutions)
                  return false;
            solution = solver.findSolution(ProblemState.RESTORE);
            if (solution == null)
                  return false;
            else
                  return true;
      }

      public Solution next() {
            solution.setSolutionNumber(++solutionNumber);
            Var[] vars = solver.getSearchStrategy().getVars();
            int[] values = new int[vars.length];
            for (int i = 0; i < values.length; i++) {
                  values[i] = solution.getValue(vars[i].getName());
            }
            try {
                  new ConstraintNotAllEqual(vars, values).post();
            } catch (Exception e) {
                  noSolutions = true;
            }
            return solution;
      }
}
```

Thus, any JSR-331 implementation may reuse common implementations or overload these methods for a better performance. However, if a JSR-331 implementation provides at least one search strategy, all other problem resolution methods can be taken from the common implementation.

More Implementation Examples

The following examples demonstrate how to apply the described Problem and Solver methods to:

- find one solution, all solutions, and an optimal solution of a simple arithmetic problem
- apply an efficient search strategy to solve the notorious Queens problem.

These problems are included in the JSR-331 Test Compatibility Kit (TCK) in the package `org.jcp.jsr331.tests`.

Simple Arithmetic Problem

This problem shares the same problem definition for different problem resolution cases.

```
//================================================
// J A V A  C O M M U N I T Y  P R O C E S S
//
// J S R  3 3 1
//
// Test Compatibility Kit
//
//================================================
package org.jcp.jsr331.tests;

import javax.constraints.Solver.Objective;
import javax.constraints.impl.Problem;
import javax.constraints.SolutionIterator;
```

```
import javax.constraints.Solver;
import javax.constraints.Var;
import javax.constraints.Solution;
import junit.framework.*;
import junit.textui.TestRunner;

public class TestSolutions extends TestCase {

	public static void main(String[] args) {
		TestRunner.run(new TestSuite(TestSolutions.class));
	}

	public Problem defineCsp() {
		Problem problem = new Problem("Test");
		//======= Define variables
		Var x = problem.variable("X", 0, 10);
		Var y = problem.variable("Y", 0, 10);
		Var z = problem.variable("Z", 0, 10);
		//======= Define constraints
		problem.post(x,"<",y);
		problem.post(y,">",5);
		problem.post(x.plus(y),"=",z);
		// Cost = 3XY - 4Z
		Var cost = x.multiply(y).multiply(3).minus(z.multiply(4));
		cost.setName("Cost");
		problem.post(cost,">=",2);
		problem.post(cost,"<=",25);
		return problem;
	}

	public void testOneSolution() {
		Problem problem = defineCsp();
		problem.log("=== One solution:");
		Solver solver = problem.getSolver();
		Solution solution = solver.findSolution();
		if (solution == null)
			problem.log("No Solutions");
		else
			solution.log();
		problem.log("After Search",problem.getVars());
		assertTrue(solution.getValue("X") == 2);
		assertTrue(solution.getValue("Y") == 6);
		assertTrue(solution.getValue("Z") == 8);
		assertTrue("testOneSolution: Invalid Cost",
				solution.getValue("Cost") == 4);
	}

	public void testAllSolutions() {
		Problem problem = defineCsp();
		problem.log("=== All solutions:");
		Solver solver = problem.getSolver();
		solver.setMaxNumberOfSolutions(4);
		Solution[] solutions = solver.findAllSolutions();
		for(Solution sol : solutions) {
			sol.log();
		}
		assertTrue(solutions.length == 4);
	}

	public void testSolutionIterator() {
		Problem problem = defineCsp();
		problem.log("=== Solution Iterator:");
		Solver solver = problem.getSolver();
```

```
        SolutionIterator iter = solver.solutionIterator();
        int n = 0;
        while(iter.hasNext()) {
            Solution solution = iter.next();
            solution.log();
            n++;
        }
        assertTrue(n == 5);
    }

    public void testOptimalSolution() {

        Problem problem = defineCsp();
        problem.log("=== Optimal Solution:");
        Solver solver = problem.getSolver();
        Var costVar = problem.getVar("Cost");
        Solution solution =
                solver.findOptimalSolution(Objective.MAXIMIZE, costVar);
        if (solution == null)
            problem.log("No Solutions");
        else
            solution.log();
        problem.log("Cost=" + solution.getValue("Cost"));
        assertTrue(solution.getValue("Cost") == 23);
    }
}
```

Queens Problem

The eight-queens problem is a well known problem that involves placing eight queens on a chess board in such a way that none of them can capture any other using the conventional moves allowed to a queen.

```
package org.jcp.jsr331.samples;

import javax.constraints.SearchStrategy;
import javax.constraints.Solution;
import javax.constraints.Solver;
import javax.constraints.Var;
import javax.constraints.impl.Problem;

public class Queens extends Problem {

    int size;
    Var[] x;

    public Queens(int size) {
        this.size = size;
    }

    public void define() {
        log("Queens " + size + ". ");
        // create 3 arrays of variables
        x = variableArray("x",0, size-1, size);
        Var[] x1 = new Var[size];
        Var[] x2 = new Var[size];
        for (int i = 0; i < size; i++) {
            x1[i] = x[i].plus(i);
            x2[i] = x[i].minus(i);
        }
```

```
        // post "all different" constraints
        postAllDifferent(x);
        postAllDifferent(x1);
        postAllDifferent(x2);
    }

    public void solve() {

        //========= Problem Resolution ==================
        // Find a solution
        Solver solver = getSolver();
        solver.setTimeLimit(600000); // milliseconds
        SearchStrategy strategy = solver.getSearchStrategy();
        strategy.setVars(x);
        strategy.setVarSelectorType(VarSelectorType.MIN_DOMAIN_MIN_VALUE);
        strategy.setValueSelectorType(ValueSelectorType.MIN);
        Solution solution = solver.findSolution();
        if(solution == null)
               log("no solutions found");
        else{
               solution.log();
        }
        solver.logStats();
    }

    public static void main(String[] args) {
        String arg = (args.length == 0) ? "1000" : args[0];
        int n = Integer.parseInt(arg);
        Queens p = new Queens(n);
        p.define();
        p.solve();
    }
}
```

An initial JSR-331 test implementation produced the following results:

```
Queens 1000
Solution #1: x-0[0] x-1[555] x-2[1] x-3[502] x-4[2] x-5[507] …
*** Execution Profile ***
Number of Choice Points: 996
Number of Failures: 8
Execution time: 1093 msec
```