WIT-J
Watson Implosion Technology - Java Interface
Application Developer's Guide

Robert Wittrock
Department of Business Analytics and Mathematical Science
IBM T.J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

January 4, 2013

Table of Contents

1. Introduction

This document describes WIT-J: “Watson Implosion Technology - Java Interface”. Watson Implosion Technology (WIT) is a software tool developed at IBM Research that solves a class of resource-constrained planning problems that may be formulated as production planning problems called “implosion” problems. For a detailed introduction to WIT, see Wittrock (2006). A comprehensive user's guide and reference manual (IBM, 2009) is also available. The present document assumes that the reader is familiar with WIT.

WIT-J is a new interface to WIT. As explained in the WIT documentation, WIT is applied to practical problems by building an application program which accesses WIT's modeling and solving capabilities by making function calls to WIT's Application Program Interface (API). WIT-J functions as a wrapper around WIT's API and provides an alternative means of accessing WIT's capabilities. There are two essential aspects of WIT-J that distinguish it from WIT's own API:

- WIT's API is in C (and a little C++). An application program that uses it must be written in either C or C++. WIT-J is in Java. An application program that uses it must be written in Java.
- While WIT is conceptually object-oriented, WIT's API is actually quite procedural. WIT-J is a completely object-oriented interface.

Thus WIT-J enables WIT application programs to be written in Java, accessing WIT's capabilities in an object-oriented style. Such programs are called WIT-J application programs.

This document is primarily addressed to anyone wishing to write WIT-J application programs and is intended to provide the information necessary to do so, when used in conjunction with the documentation on WIT itself. It is intended to function both as a tutorial on WIT-J and as a reference manual:

- Sections 1 through 18 are structured to be read from beginning to end, so as to be useful as a tutorial.
- Section 19 gives an example of a short WIT-J application program. This example is referred to numerous times throughout the main text.
- Sections 20 through 22 provide comprehensive lists and tables. Sections 21 through 22 contain many cross references to the preceding sections of this document so as to facilitate look up, as in a reference manual.

Availability: WIT-J uses Java 6.0. It's currently available on Linux and on Windows. Ports to other platforms will be created as needed.

2. Architecture

To begin, a basic understanding of WIT-J's high-level architecture will be useful. The WIT-J code consists of the following three portions:

- The Java portion ("WIT-J/Java"): A collection of classes in Java.
- The C++ portion ("WIT-J/C++"): A collection of classes and global functions in C++.
- WIT itself.

The Java portion of WIT-J has the following responsibilities:

- Its public classes, methods, and fields constitute WIT-J's API.
- It maintains an object-oriented representation of the implosion problem defined in WIT. Specifically, it stores a representation of the structure of the problem, but not the numerical data.
- It interacts with the C++ portion of WIT-J.

The C++ portion of WIT-J has the following responsibilities:

- Interacts with the Java portion of WIT-J.
- It maintains an object-oriented structural representation of the implosion problem similar to the representation in the Java portion.
- It translates between WIT-J's object-oriented representation of the implosion problem and the more procedural representation required by WIT's API.
- It makes the direct calls to WIT's API functions.

The C++ portion is completely internal to WIT-J and not visible to the application program.

The interaction between the Java portion and the C++ portion of WIT-J is made possible by using JNI, the Java Native Interface. JNI consists of a collection of functions and data types in C and C++ that enable Java methods that are declared `native` to be implemented in C or C++ and enable code written in C or C++ to access code written in Java. It is included as part of the Java Virtual Machine implementation on any given platform. For complete information on JNI, see Liang (1999).

The execution of a typical WIT-J method might proceed as follows:

- The WIT-J application program invokes a public method of the Java portion of WIT-J.
- The public WIT-J/Java method performs error checking, translates its arguments into arguments more suitable for a native method invocation (if necessary), and then passes them to a corresponding package-internal native method in WIT-J/Java.
- JNI fulfills the call to the Java native method as a call to a corresponding global function in WIT-J/C++, called a "native implementation function".
- The native implementation function uses JNI functions to translate JNI's C/C++ representation of the arguments to the Java native method into ordinary C++ objects and primitives and then passes them to a corresponding member function of a WIT-J/C++ class.
- The WIT-J/C++ member function translates its object-oriented arguments into the style of argument required for a call to a WIT API function, and then passes them to the appropriate WIT API function.
- The WIT API function executes.

Note that not all WIT-J methods work in this way; for example, some methods do not invoke WIT-J/C++ at all.

3. Class Overview

The classes that constitute the Java portion of WIT-J all belong to the following package:

```
com.ibm.witj
```

Thus it may be helpful to include the following statement in any source file that makes much use of WIT-J:

```
import com.ibm.witj.*;
```

Alternatively, in Eclipse the appropriate single-type-import statements can be generated automatically. The code examples in this guide will assume that the source code contains the appropriate import statement(s).

The most important public classes in WIT-J's Java portion are the following:

`Problem:`
A `Problem` represents an implosion problem. It corresponds to a `WitRun` in WIT.

`Part:`
A `Part` represents a part in an implosion problem.

`Demand:`
A `Demand` represents a demand in an implosion problem.

`Operation:`
An `Operation` represents an operation in an implosion problem.

`BomEntry:`
A `BomEntry` represents a BOM entry in an implosion problem.

`Substitute:`
A `Substitute` represents a substitute BOM entry in an implosion problem.

`BopEntry:`
A `BopEntry` represents a BOP entry in an implosion problem.

`Component.`
This is a superclass of classes `Part`, `Demand`, `Operation`, `BomEntry`, `Substitute`, `BopEntry`, and `Problem`. A `Component` represents a “WIT data object”, also called a “WIT component”. Note that a `Problem` is considered to be a `Component` of itself.

`Attribute <V>:`
An `Attribute <V>` represents a WIT data attribute whose values are of type `V`.

These classes (and others) will be further explained in the remainder of this document.

4. Factory Methods

WIT-J classes don't have public constructors. Instead, if a class in WIT-J allows instantiation by the application program, it provides a public factory method: a static method that constructs a new instance of the class and returns it. Factory methods in WIT-J are always called `newInstance`. Note however that some public classes in WIT-J do not provide a public factory method: their instances are constructed as side-effects of methods of other classes or as an aspect of static initialization. The main rationale for providing factory methods in WIT-J instead of constructors is to allow error checking to be completed before construction begins. See Bloch (2008), items 1 and 64.

The following are the public factory methods for the classes discussed so far:

Class: **`Problem`**
Method: `static Problem` **`newInstance`** `()`

Constructs and returns a new `Problem`. Constructing a Problem causes various set-up activities to be performed; most importantly, it causes the C++ portion of WIT-J to construct a new `WitRun`, which is called the “underlying `WitRun`” of the `Problem`. Specifically, it invokes the WIT API functions `witNewRun` and `witInitialize` on the underlying `WitRun`. (The calls to `witNewRun` and `witInitialize` are “silent”: WIT will not issue any informational messages for them.)

Class: **`Part`**
Method:

```
static Part newInstance (
   Problem       theProblem,
   String        thePartName,
   Part.Category theCategory)
```

Constructs and returns a new `Part`. The new `Part` will:

- Belong to `theProblem`.
- Have a part name given by `thePartName`.
- Belong to the part category identified by `theCategory`, which is an instance of the enum `Part.Category`. This is an `enum` that is statically nested in class `Part` and its instances identify the part category of a part. It has two constants: `MATERIAL` and `CAPACITY`.

Class: **`Demand`**
Method: static Demand **`newInstance`** `(Part thePart, String theDemandName)`

Constructs and returns a new `Demand`. The new `Demand` will:

- Belong to the same `Problem` as `thePart`.
- Be a `Demand` for `thePart`.
- Have a demand name given by `theDemandName`.

Class: **`Operation`**
Method:
```
static Operation newInstance (
   Problem theProblem,
   String  theOperationName)
```

Constructs and returns a new `Operation`. The new `Operation` will:

- Belong to `theProblem`.
- Have an operation name given by `theOperationName`.

Class: **`BomEntry`**
Method: `static BomEntry newInstance (Operation theOpn, Part thePart)`

Constructs and returns a new `BomEntry`. Verifies that `theOpn` and `thePart` belong to the same `Problem`. The new `BomEntry` will:

- Belong to the same `Problem` as `theOpn` and `thePart.`
- Have `theOpn` as its consuming `Operation`.
- Have `thePart` as its consumed `Part`.

Class: **`Substitute`**
Method: `static Substitute newInstance (BomEntry theBomEnt, Part thePart)`

Constructs a new `Substitute` and returns it. Verifies that `theBomEnt` and `thePart` belong to the same `Problem`. The new `Substitute` will:

- Belong to the same `Problem` as `theBomEnt` and `thePart.`
- Have `theBomEnt` as its replaced `BomEntry`.
- Have `thePart` as its consumed `Part`.

Class: **`BopEntry`**
Method: `static BopEntry newInstance (Operation theOpn, Part thePart)`

Constructs a new `BopEntry` and returns it. Verifies that `theOpn` and `thePart` belong to the same `Problem`. The returned `BopEntry` will:

- Belong to the same `Problem` as `theOpn` and `thePart.`
- Have `theOpn` as its producing `Operation`.
- Have `thePart` as its produced `Part`.

Ground Rule:

Except where otherwise indicated, all reference arguments to WIT-J methods are required to be non-`null` ; if a reference argument is `null`, an exception is thrown. So in the case of factory methods, all arguments must be non-`null` .

Usage Advice:

As mentioned, a factory method returns a reference to the new instance that it has created. An application program might typically store this reference in any of the following ways:

- It might store the reference in a local variable.
- It might store the reference in a field of an object in the application program.
- It might store the reference as a value in a `HashMap`.
- It might simply ignore the reference: WIT-J stores its owns reference to each `Component` that it creates. These references can be retrieved at any time, using WIT-J's navigation methods. (See Section 5.)

Code Example:

See Section 19 on page 41.

5. Navigation Methods

WIT-J provides a variety of "navigation methods": methods that allow the application program to progress easily between each of the Components of a Problem and the Problem itself. The following are WIT-J's navigation methods:

Each of the following methods returns a List of all the instances of a particular class that are associated with the object on which the method was called. For example, let theProblem be a Problem. Then theProblem.getParts () returns a List of all Parts associated with the Problem. In each case, the List returned is actually an unmodifiable view of an internal List. The objects in each returned List are listed in the order in which they were created.

```
Class:   Problem
Method:  List <Component>  getComponents  ()
Method:  List <Part>       getParts       ()
Method:  List <Demand>     getDemands     ()
Method:  List <Operation>  getOperations  ()
Method:  List <BomEntry>   getBomEntries  ()
Method:  List <Substitute> getSubstitutes ()
Method:  List <BopEntry>   getBopEntries  ()
```

```
Class:   Part
Method:  List <Demand>     getDemands     ()
Method:  List <BomEntry>   getBomEntries  ()
Method:  List <Substitute> getSubstitutes ()
Method:  List <BopEntry>   getBopEntries  ()
```

```
Class:   Operation
Method:  List <BomEntry>   getBomEntries  ()
Method:  List <BopEntry>   getBopEntries  ()
```

```
Class:   BomEntry
Method:  List <Substitute> getSubstitutes ()
```

Code Example:

See Section 19 on page 41. (getPart for class Demand)

Each of the following methods returns the unique instance of a particular Component class that is associated with the object on which the method was called and is identified by the specified name. If no such instance exists, null is returned. For example, let theProblem be a Problem and thePartName be a String. Then theProblem.getPart (thePartName) returns the Part associated with theProblem and whose partName matches thePartName, or null, if there is no such Part.

```
Class:   Problem
Method:  Part      getPart      (String thePartName)
Method:  Operation getOperation (String theOperationName)
```

```
Class:   Part
Method:  Demand    getDemand    (String theDemandName)
```

Each of the following methods returns the unique instance of a particular class that's associated with the object on which the method was called. For example, let `theBomEntry` be a `BomEntry`. Then `theBomEntry.getPart ()` returns the consumed `Part` for `theBomEntry`.

Class: **Component**
Method: `Problem` **`getProblem`** `()`

Class: **Demand**
Method: `Part` **`getPart`** `()`

Class: **BomEntry**
Method: `Part` **`getPart`** `()`
Method: `Operation` **`getOperation`** `()`

Class: **Substitute**
Method: `Part` **`getPart`** `()`
Method: `BomEntry` **`getBomEntry`** `()`

Class: **BopEntry**
Method: `Part` **`getPart`** `()`
Method: `Operation` **`getOperation`** `()`

Code Example:

See Section 19 on page 41. (`getPart` for class `Demand`)

Each of the following methods returns the number of instances of a particular `Component` class that have been created in the `Problem` on which the method was called. For example, `getNPartsCreated ()` returns the number of `Parts` that have been created in the `Problem`.

Class: **Problem**
Method: `int` **`getNPartsCreated`** `()`
Method: `int` **`getNDemandsCreated`** `()`
Method: `int` **`getNOperationsCreated`** `()`

Note that if a `Component` has been deactivated by the `purgeData` method (see Section 15 on page 36), it is still included in this count.

The count returned by each of these methods can be used when computing the names of new instances of the `Component` class so as to ensure that each instance has a distinct name. For example, if each `Part` in a `Problem` is given the name computed by the following expression, then each `Part` will have a distinct name:

```
"Widget #" + theProblem.getNPartsCreated ()
```

Each of the following methods returns a `List` of all instances of a particular `Component` class that are associated with both the object on which the method was called and the `Part` argument. For example, let `theOpn` be an `Operation` and `thePart` be a `Part`. Then `theOpn.getBomEntriesTo (thePart)` returns a `List` of all `BomEntries` connecting the `theOpn` to `thePart.` In each case, the `List` returned is newly created when the method is called and the objects it contains are listed in the order in which they were created.

Class: **`Operation`**
Method: `List <BomEntry>` **`getBomEntriesTo`** `(Part thePart)`
Method: `List <BopEntry>` **`getBopEntriesTo`** `(Part thePart)`

Class: **`BomEntry`**
Method: `List <Substitute>` **`getSubstitutesTo`** `(Part thePart)`

Consider the following three methods:

Class: **`Operation`**
Method: `BomEntry` **`getUniqueBomEntryTo`** `(Part thePart)`
Method: `BopEntry` **`getUniqueBopEntryTo`** `(Part thePart)`

Class: **`BomEntry`**
Method: Substitute **`getUniqueSubstituteTo`** `(Part thePart)`

In many cases, one can determine in advance (by the structure of the application's WIT model) that there is at most one `BomEntry` connecting a particular `Operation` to a particular `Part`. In such a case, `getUniqueBomEntryTo` can be used. Let `theOpn` be an `Operation` and `thePart` be a `Part`. Then `theOpn.getUniqueBomEntryTo (thePart)` behaves as follows:

- If there is exactly one `BomEntry` connecting `theOpn` to `thePart`, that `BomEntry` is returned.
- If there is no `BomEntry` connecting `theOpn` to `thePart`, `null` is returned.
- If there is more than one `BomEntry` connecting `theOpn` to `thePart`, an exception is thrown.

The other methods listed above work analogously.

6. Introduction to Attributes

In general, the non-structural data for an implosion problem in WIT is specified by attributes of the components. In WIT-J, the attributes of an implosion problem are (with a few exceptions) represented by instances of a generic class: `Attribute <V>`. The type argument, `V`, specifies the "value type" of the `Attribute`, i.e., the type of value that is stored for the `Attribute`. For example, consider the WIT attribute buildNstn. The corresponding WIT-J `Attribute` is an object of type `Attribute <Boolean>` because the value of the corresponding WIT attribute is of type boolean.

The `Attribute` objects in WIT-J are not constructed by a factory method; they are automatically constructed as an aspect of the static initialization of the `Attribute` class. For each `Attribute`, there is a `public static final` field of class `Attribute`, whose name matches that of the `Attribute` and whose value is the `Attribute`.

The `Attribute` objects in WIT-J are identified by a somewhat different naming than the one used in WIT:

- All letters in the name of a WIT-J `Attribute` are upper case.
- For each upper case letter in the name of a WIT attribute, the corresponding letter in the name of the WIT-J `Attribute` is preceded by an underscore, "_".
- In all other respects, the name of a WIT-J `Attribute` will match the name of the WIT attribute that it represents.

This naming convention is used for WIT-J `Attributes` in order to adhere to standard practice for the names of constant fields in Java.

For example:

Class: **`Attribute`**
Field: `static final Attribute <Boolean>` **`BUILD_NSTN`**

The field `Attribute.BUILD_NSTN` stores the WIT-J `Attribute` that represents the WIT attribute buildNstn.

For convenience, it may be helpful to include the following import statement in any source file that makes much use of WIT-J `Attributes`:

```
import static com.ibm.witj.Attribute.*;
```

The code examples in this guide will assume that the source file contains this import statement.

There are currently 148 `Attributes` in WIT-J. They are listed in Table 1 on page 48.

Each `Attribute` is said to "apply to" one or more subclasses of class `Component`: Specifically, an `Attribute`, `theAtt` applies to a `Component` class, `theClass`, iff each instance of `theClass` stores a value for `theAtt`. For example, the `OFFSET Attribute` applies to classes `BomEntry`, `Substitute` and `BopEntry`. Table 1 on page 48 shows the set of `Component` classes to which each `Attribute` applies. An `Attribute` applies to a specific `Component` (instance), iff it applies to the class to which the `Component` belongs. Global `Attributes` apply to class `Problem`.

The following are all the types that are used as value types for WIT-J `Attributes`:

- `Boolean`
- `Integer`
- `Double`
- `boolean[]`
- `int[]`
- `double[]`
- `String`
- `BoundSet`

Notes on `Attribute` value types:

- Table 1 on page 48 shows the value type of each `Attribute`.
- When the value type an an `Attribute` is an array type, the length of the array stored for the `Attribute` must match the number of periods in the corresponding `Problem`.
- Note that there are no value types based on `float`, but there are value types based on `double`. This is an instance of a more general principle: WIT-J never uses types based on `float`; instead, it uses types based on `double`. Furthermore, when a WIT API function uses a type based on `double` in its argument list, the suffix "`Dbl`" is appended to the end of the name of the API function. WIT-J does not follow this convention. Nothing is appended to the end of a method name when the method uses a type based on `double` in its argument list.
- The class `BoundSet` listed above is a vacuous class in WIT-J: It has no public methods at all. It is only used to create the parameterized type `Attribute <BoundSet>`, which represents any WIT attribute whose value type is (conceptually) a Bound Set.

Finally, note that `Attributes` are immutable objects: All state information associated with an `Attribute` is determined during the construction of the `Attribute` and not changed after that.

7. Methods for Setting Attribute Values

WIT-J provides a number of methods that enable the application program to set the value of a WIT attribute represented by a WIT-J `Attribute` for a specific WIT component represented by a WIT-J `Component`. This is called "setting the value of an `Attribute` for a `Component`".

The first value-setting method to consider is the generic `set` method:

Class: **`Component`**
Method: `<V> void` **`set`** `(Attribute <V> theAtt, V theValue)`

- Sets the value of the `Attribute` for the `Component` to `theValue`.
- The value type of the `Attribute` can be any value type except `BoundSet`. If the value type is `BoundSet`, an exception is thrown.
- If the `Attribute` does not apply to the `Component`, an exception is thrown. Table 1 on page 48 shows the set of `Component` classes to which each `Attribute` applies.
- An `Attribute` is said to be modifiable, iff its value can be set by the application program. Table 1 on page 48 indicates which `Attributes` are modifiable. If the `set` method is invoked on a non-modifiable `Attribute`, an exception is thrown.
- If the value type is an array type and the length of the array passed as `theValue` does match the number of periods in the `Problem` associated with the `Component`, an exception is thrown.

<u>Code Example:</u>

See Section 19 on page 41.

For each of the array-valued `Attribute` types, a `setVectorToScalar` method is provided:

Class: **`Component`**

```
Method: void setVectorToScalar (Attribute <boolean[]> theAtt,
                                boolean               theValue)
Method: void setVectorToScalar (Attribute <int[]>     theAtt,
                                int                   theValue)
Method: void setVectorToScalar (Attribute <double[]>  theAtt,
                                double                theValue)
```

- Each of these methods sets the value of each vector element of the `Attribute` for the `Component` to `theValue`.
- If the `Attribute` does not apply to the `Component` or is not modifiable, an exception is thrown.

<u>Code Example:</u>

See Section 19 on page 41.

For `Attributes` of value type `BoundSet`, the following methods are provided:

Class: **`Component`**
Method:
```
void setBoundSet (Attribute <BoundSet>  theAtt, double[]  hardLBArr,
                                                double[]  softLBArr,
                                                double[]  hardUBArr)
```

- Sets the values of the three bound vectors of the `Attribute` for the `Component` to the three arrays passed.
- If the `Attribute` does not apply to the `Component`, an exception is thrown.
- If the length of any of the argument arrays does match the number of periods in the `Problem` associated with the `Component`, an exception is thrown.
- The array arguments may be `null` references. If a `null` reference is passed, the corresponding bound vector of the represented WIT attribute is left unchanged.

Class: **`Component`**
Method:
```
void setBoundSetToScalars (Attribute <BoundSet> theAtt,
                           double               hardLBVal,
                           double               softLBVal,
                           double               hardUBVal)
```

- Sets the value of each vector element of each of the three bound vectors of the `Attribute` for the `Component` to the corresponding scalar argument.
- If the `Attribute` does not apply to the `Component`, an exception is thrown.

<u>Code Example:</u>

Let `theOpn` be an Operation:

```
theOpn.setBoundSetToScalars (execBounds, 0.0, 100.0, 500.0);
```

This sets the value of the execBounds attribute for `theOpn` to (0, 100, 500) in every period.

8. Methods for Retrieving Attribute Values

WIT-J provides a number of methods that enable the application program to retrieve the value of a WIT attribute represented by a WIT-J `Attribute` for a specific WIT component represented by a WIT-J `Component`. This is called "retrieving the value of an `Attribute` for a `Component`".

The first value-retrieving method to consider is the generic `get` method:

Class: **`Component`**
Method: `<V> V` **`get`** `(Attribute <V> theAtt)`

- Returns the value of the `Attribute` for the `Component`.
- The value returned is a newly created instance of type `V`.
- If the value type of the `Attribute` is `BoundSet`, an exception is thrown.
- If the `Attribute` does not apply to the `Component`, an exception is thrown.

Code Example:

See Section 19 on page 41.

Another kind of value-retrieving method is for array-valued Attributes only:

Class: **`Component`**
Method: `<V> void` **`getVector`** `(Attribute <V> theAtt, V theArray)`

- Copies the value of the `Attribute` for the `Component` into `theArray.`
- If the value type of the `Attribute` is not `boolean[]`, `int[]`, or `double[]`, an exception is thrown.
- If `theArray.length` does not match the number of periods in the `Problem` associated with the `Component`, an exception is thrown.
- If the `Attribute` does not apply to the `Component`, an exception is thrown.

Code Example:

Let `theProblem` be a `Problem`:

```
double[] execVolArr;

execVolArr = new double[theProblem.get (N_PERIODS)];

for (Operation theOpn: theProblem.getOperations ())
   {
   theOpn.getVector (EXEC_VOL, execVolArr);

   processExecVol (execVolArr);
   }
```

This code fragment iterates through each `Operation` in `theProblem`, retrieves the value of the execVol `Attribute` for the `Operation`, and passes it to a method called `processExecVol`.

Finally, there's a value-retrieving method for `BoundSet`-valued `Attributes`:

Class: **`Component`**
Method:

```
void getBoundSet (Attribute <BoundSet> theAtt, double[] hardLBArr,
                                               double[] softLBArr,
                                               double[] hardUBArr)
```

- Copies the value of each bound vector of the `Attribute` for the `Component` into the corresponding array argument.
- If the length of any array argument does not match the number of periods in the `Problem` associated with the `Component`, an exception is thrown.
- If the `Attribute` does not apply to the `Component`, an exception is thrown.

9. Methods for Working with Attributes

This section describes WIT-J's methods for working with the `Attributes` themselves (at the "meta" level). Note, however, that many application programs will not need to interact with `Attributes` in this way, and so it should be harmless to skim or skip this section on first reading.

Class: **`Attribute`**
Method: `static List <Attribute <?>>` **`getAttributes`** `()`

- Returns an unmodifiable `List` of all `Attributes` in WIT-J.

Code Example:

See "Code Example for `Attribute` Methods" on page 21.

Class: **`Attribute <V>`**
Method: `Class <V>` **`getValueType`** `()`

- Returns the value type of the `Attribute`.

Code Example:

See "Code Example for `Attribute` Methods" on page 21.

Class: **`Attribute <V>`**
Method: `<V2> Attribute <V2>` **`asAttOfType`** `(Class <V2> theValueType)`

- Returns the `Attribute`, converted to an `Attribute <V2>`, where type `V2` is identified by `theValueType`.
- If `theValueType` is not the value type of the `Attribute`, an exception is thrown.
- This method would normally be used with respect to a variable of type `Attribute <?>,` as would be obtained by a call to `Attribute.getAttributes ().`
- This method is intended to be used in place of an explicit cast to type `Attribute <V2>`, which would cause the compiler to issue an "unchecked cast" warning. This method does the required type checking.

Code Example:

See "Code Example for `Attribute` Methods" on page 21.

Class: **`Attribute <V>`**
Method: `boolean` **`appliesTo`** `(Class <? extends Component> theClass)`

- Returns true, iff the `Attribute` applies to `theClass`.
- `theClass` should be the `Class` object for a subclass of class `Component`; if it's `Component.class`, an exception is thrown.

Class: **`Attribute <V>`**
Method: `boolean` **`isModifiable`** `()`

- Returns true, iff the `Attribute` is modifiable, i.e., its value can be set by the application program.

Class: **`Attribute <V>`**
Method: `String` **`toString`** `()`

- Override from class `Object`.
- Returns the name of the `Attribute`, i.e., its "WIT-J name".
- For example, `BUILD_NSTN.toString ()` returns `"BUILD_NSTN"`.

Code Example:

See "Code Example for `Attribute` Methods" on page 21.

Class: **`Attribute <V>`**
Method: `String` **`getWitName`** `()`

- Returns the name of WIT attribute represented by the `Attribute`, i.e., its "WIT name".
- For example, `BUILD_NSTN.getWitName ()` returns `"buildNstn"`.

Class: **`Attribute <V>`**
Method: `boolean` **`isValidFor`** `(Component theComp)`

- Returns true, iff the `Attribute` is valid for a call to the `get` method on `theComp` in the present state. If it is not valid, it may be for any of the following reasons:
  - The `Attribute` does not apply to the class to which the `theComp` belongs.
  - The `Attribute` is only valid for `Parts` of part category "material" and `theComp` is a `Part` of part category "capacity".
  - The `Attribute` is only valid when the `Problem` is in stochastic mode and `theComp` belongs to a `Problem` that is not in stochastic mode.
  - The `Attribute` is only valid when the `Problem` is in multiple objectives mode and `theComp` belongs to a `Problem` that is not in multiple objectives mode.

Code Example:

See "Code Example for `Attribute` Methods" on page 21.

Class: **`Attribute <V>`**
Method: `boolean` **`hasDefaultValue`** `()`

- Returns true, iff the `Attribute` has a default value. For example, `PART_NAME` has no default value.

Code Example:

See “Code Example for `Attribute` Methods” on page 21.

Class: **`Attribute <V>`**
Method: `V` **`getDefaultValue`** `(Problem theProblem)`

- Returns the default value of the `Attribute` in the context of `theProblem`.
- Note that sometimes the default value of a WIT attribute depends on the WIT problem in which the attribute is to be considered: The default value might be nPeriods – 1 or a vector of length nPeriods.
- If the value type of the `Attribute` is `BoundSet`, an exception is thrown.
- If the `Attribute` doesn't have a default value, an exception is thrown. See the method `hasDefaultValue.`

Code Example:

See “Code Example for `Attribute` Methods” on page 21.

Class: **`Attribute <V>`**
Method:

```
void getDefaultBoundSet (Problem  theProblem,
                         double[] hardLBArr,
                         double[] softLBArr,
                         double[] hardUBArr)
```

- Copies the default values of the `Attribute` into the three array arguments.
- If the value type of the `Attribute` is not `BoundSet`, an exception is thrown.
- If the length of any of the arrays does not match the number of periods in the `Problem`, an exception is thrown.

Code Example for `Attribute` Methods:

The following method, `displayNonDefGlobalBoolAtts (theProblem)`, illustrates many of the `Attribute` methods described in this section. It proceeds as follows: For each global boolean `Attribute`, if the `Attribute` is currently valid but not at its default value in `theProblem`, the method displays the `Attribute`, its WIT name, its default value, and its current value.

```
static void displayNonDefGlobalBoolAtts (Problem theProblem)
   {
   Attribute <Boolean> theBoolAtt;
   boolean             defValue;
   boolean             theValue;

   for (Attribute <?> theAtt: getAttributes ())
      {
      if (theAtt.isValidFor (theProblem))
         {
         if (theAtt.getValueType () == Boolean.class)
            {
            theBoolAtt = theAtt.asAttOfType (Boolean.class);

            if (theBoolAtt.hasDefaultValue ())
               {
               defValue = theBoolAtt.getDefaultValue (theProblem);

               theValue = theProblem.get (theBoolAtt);

               if (theValue != defValue)
                  {
                  System.out.printf (
                       "%n"
                     + "Non-default global boolean Attribute found.%n"
                     + "%n"
                     + "   Attribute:     %s%n"
                     + "   WIT Name:      %s%n"
                     + "   Default Value: %s%n"
                     + "   Current Value: %s%n",
                     theBoolAtt.toString (),
                     theBoolAtt.getWitName (),
                     defValue,
                     theValue);
                  }
               }
            }
         }
      }
   }
```

10. Wrapper Methods

Consider the following method:

Class: `Problem`
Method: `void` **`optImplode`** `()`

This method performs optimizing implosion on the `Problem`. It executes by causing WIT-J/C++ to invoke `witOptImplode (...)` on the underlying `WitRun` of the `Problem`.

`optImplode ()` is an example of a "wrapper method" in WIT-J. In general, a wrapper method will have the following characteristics:

- It corresponds to a WIT API function, which is said to be "wrapped" by the wrapper method.
- Its execution involves an invocation of the wrapped WIT API function.
- The name of the wrapped WIT API function is similar to that of the wrapper method. In most cases, the wrapped WIT API function will have the "corresponding canonical name", which can be obtained by starting with the name of the wrapper method, converting the first letter to upper case, and appending "`wit`" to the beginning, so that, e.g., `witOptImplode` is the corresponding canonical name for `optImplode`.
- Its arguments will correspond logically to the arguments of the wrapped WIT API function, but translated into WIT-J style arguments. Thus, while `witOptImplode` takes a single argument of type `WitRun *`, `optImplode` takes no arguments, but must be invoked on a WIT-J `Problem`, which corresponds to a `WitRun`.

Each of the following wrapper methods is a method of class `Problem`, takes no arguments, and wraps a WIT API function that has the corresponding canonical name and takes a pointer to the underlying `WitRun` as its only argument:

Class: **`Problem`**
Method: `void` **`clearCplexParSpecs`** `()`
Method: `void` **`clearStochSoln`** `()`
Method: `void` **`evalObjectives`** `()`
Method: `void` **`finishHeurAlloc`** `()`
Method: `void` **`generatePriorities`** `()`
Method: `void` **`heurImplode`** `()`
Method: `void` **`mrp`** `()`
Method: `void` **`optImplode`** `()`
Method: `void` **`postprocess`** `()`
Method: `void` **`preprocess`** `()`
Method: `void` **`shutDownHeurAlloc`** `()`
Method: `void` **`startHeurAlloc`** `()`
Method: `void` **`stochImplode`** `()`

Note that some of WIT's main capabilities are invoked by these methods: heuristic implosion, optimizing implosion, and stochastic implosion.

Code Example:

See Section 19 on page 41. (`heurImplode`)

A second set of wrapper methods in WIT-J are the file-writing methods. Each of these methods takes a `String` argument indicating a file name. If the `fileName String` is the empty `String`, the current message file is used. Some of the methods also take an argument that's an instance of `enum FileFormat`. This `enum` indicates a file format to be used and has two constants: `BSV` (for Blank Separated Values) and `CSV` (for Comma Separated Values). Each file-writing method wraps a WIT API function with the corresponding canonical name.

The following are WIT-J's file-writing methods:

```
Class:   Problem
Method:  void displayData       (String fileName)
Method:  void writeCriticalList (String fileName, FileFormat theFormat,
                                                  int        maxListLen)
Method:  void writeData         (String fileName)
Method:  void writeExecSched    (String fileName, FileFormat theFormat)
Method:  void writeReqSched     (String fileName, FileFormat theFormat)
Method:  void writeShipSched    (String fileName, FileFormat theFormat)
```

Consider the following wrapper method:

```
Class:   Problem
Method:  void readData (String fileName)
```

- This method is a wrapper for `witReadData`, which reads the WIT data file identified by `fileName` into the `WitRun`.
- May only be called on an “empty” `Problem`: If the `Problem` has one or more `Parts` or `Operations`, an exception is thrown.
- For each WIT component (i.e., part, BOM entry, etc.) that has been read into the underlying `WitRun`, a corresponding `Component` is constructed in WIT-J and associated with the `Problem`.
- The wit34Compatible attribute must not be set in the data file. If it is set, a `TerminalAppException` exception is thrown. (See Section 17, “Exceptions” on page 39.)
- The characters in the data file must be in UTF-8 format. If a non-UTF-8 character is read, a `BadCharacterException` exception is thrown. (See Section 17, “Exceptions” on page 39.)

Consider the following wrapper method:

Class: **Component**
Method: void **copyComponentData** (Component origComp)

- This method invokes WIT to copy the input data from origComp into this Component.
- This method is a wrapper for all of the following WIT API Functions:
  - witCopyPartData
  - witCopyDemandData
  - witCopyOperationData
  - witCopyBomEntryData
  - witCopySubsBomEntryData
  - witCopyBopEntryData
- The origComp argument must be an instance of the same class as this Component. For example, if copyComponentData is called on a Part, then the origComp argument must be a Part.
- copyComponentData must not be called on a Problem.
- If this Component and origComp are the same Object, no action is taken.

Consider the following wrapper method:

Class: **Demand**
Method: double **incHeurAlloc** (int shipPeriod, double desIncVol)

- This method is a wrapper for witIncHeurAllocDbl.
- The value returned is the incremental shipVol achieved by the method.

Consider the following wrapper method:

Class: **Problem**
Method:

```
void eqHeurAlloc (
         ArrayList <Demand>  theDemandList,
         ArrayList <Integer> shipPeriodList,
         ArrayList <Double>  desIncVolList,
         ArrayList <Double>  incVolList)
```

- This method is a wrapper for witEqHeurAllocDbl.
- The first three ArrayList arguments should all have the same size.
- For 0 ≤ theIdx < theDemandList.size (), theDemandList.get (theIdx), shipPeriodList.get (theIdx), and desIncVolList.get (theIdx) specify the allocation targets for theIdx.
- On return, for 0 ≤ theIdx < theDemandList.size (), incVolList.get (theIdx) is the incremental shipment volume achieved for theIdx.

Consider the following wrapper method:

Class: **Problem**
Method: void **getCriticalList** (

```
ArrayList <Part>    critPartList,
ArrayList <Integer> critPeriodList)
```

- This method is a wrapper for `witGetCriticalList`.
- On return, for $0 \leq$ `theIdx < critPartList.size ()`:

  `critPartList.get (theIdx)` is Part `#theIdx` in the critical parts list

  `critPeriodList.get (theIdx)` is period `#theIdx` in the critical parts list.

Consider the following wrapper method:

Class: **Problem**
Method: void **getPgdCritList** (

```
ArrayList <Part>    critPartList,
ArrayList <Integer> critPeriodList,
ArrayList <Demand>  theDemandList,
ArrayList <Integer> shipPeriodList)
```

- This method is a wrapper for `witGetPgdCritList`.
- On return, for $0 \leq$ `theIdx < critPartList.size ()`:

  `critPartList  .get (theIdx)` is critical Part `#theIdx` in the pegged critical list.

  `critPeriodList.get (theIdx)` is critical period `#theIdx` in the pegged critical list.

  `theDemandList .get (theIdx)` is Demand `#theIdx` in the pegged critical list.

  `shipPeriodList.get (theIdx)` is shipment period `#theIdx` in the pegged critical list.

Consider the following wrapper method:

Class: **Part**
Method: `ArrayList <Part>` **`getBelowList`** `()`

- This method is a wrapper for `witGetPartBelowList`.
- The returned `ArrayList <Part>` is the below list for the `Part`.

Consider the following wrapper methods:

Class: **Problem**
Method: `ArrayList <Part>` **`getSortedParts`** `()`
Method: `ArrayList <Operation>` **`getSortedOperations`** `()`

- This methods are wrappers for `witGetParts` and `witGetOperations`.
- Before calling the wrapped WIT API function, each of these methods calls `witPreprocess` to preprocess the data. This causes the `Parts` and `Operations` to be sorted in upward BOM order. Thus the `List` returned `getSortedParts` lists all of the `Parts` in the `Problem` in an order such that, if `Part A` is consumed by an `Operation` that produces `Part B` with an explodeable `BopEntry`, then `Part A` will appear before `Part B` in the `List`. The `List` returned by `getSortedOperations` is sorted in the same way.

Consider the following wrapper method:

Class: **Problem**
Method: void **`getExpCycle`** `(`

```
          ArrayList <Part>      thePartList,
          ArrayList <Operation> theOpnList)
```

- This method is a wrapper for `witGetExpCycle`.
- On return, for $0 \leq$ `theIdx < thePartList.size ():`

  `thePartList.get (theIdx)` is Part `#theIdx` in the retrieved explodeable cycle.

  `theOpnList .get (theIdx)` is Operation `#theIdx` in the retrieved explodeable cycle.

Consider the following wrapper methods:

Class: **`Problem`**
Method: `void` **`addIntCplexParSpec`** `(String theName, int    theValue)`
Method: `void` **`addDblCplexParSpec`** `(String theName, double theValue)`

- These methods are wrappers for `witAddIntCplexParSpec` and `witAddDblCplexParSpec`, which create CPLEX parameter specifications.

Consider the following wrapper methods:

Class: **`Problem`**
Method: `Integer` **`getIntCplexParSpec`** `(String theName)`
Method: `Double ` **`getDblCplexParSpec`** `(String theName)`

- These methods are wrappers for `witGetIntCplexParSpec` and `witGetDblCplexParSpec`, which retrieve CPLEX parameter specifications.
- If a matching CPLEX parameter specification of the indicated type exists in the **`Problem`**, its value is returned.
- If no matching CPLEX parameter specification of the indicated type exists in the **`Problem`**, a null pointer is returned.

11. Pegging Methods

WIT provides two versions of pegging: post-implosion pegging and concurrent pegging. Since the interface to both forms of pegging is similar, they are both handled with the same set of classes and methods in WIT-J.

Most of the pegging methods in WIT-J make use of the following generic class:

`class` **`PeggingTriple <C extends Component>`**

A `PeggingTriple <C>` represents a triple composed of the following three elements:

- An instance of class `C`, called the "root" of the `PeggingTriple`.
- An `int`, called the "period" of the `PeggingTriple`.
- A `double`, called the "volume" of the `PeggingTriple`.

Usually, a `PeggingTriple` will be (informally) associated with an `Attribute <double[]>` that applies to `Component` class `C`. In this case, the `PeggingTriple` will represent a portion of the value of the `Attribute` for the root in the period, where the portion is called the volume. For example, consider a `PeggingTriple <Part>` whose root is `Part` "A", whose period is 5, and whose volume is 10. This might represent 10 units of the supplyVol of `Part` "A" in period 5 that's pegged to a particular `Demand`.

The `PeggingTriple` class has the following accessor methods:

Class: **`PeggingTriple <C extends Component>`**
Method: `C` **`getRoot`** `()`
Method: `int` **`getPeriod`** `()`
Method: `double` **`getVolume`** `()`

- These methods return the root, period, and volume represented by the `PeggingTriple`.

The `PeggingTriple` class also has the following method, overridden from class `Object`:

Method: `String` **`toString`** `()`

The `PeggingTriple` class does not have a `newInstance` method; instances of it can only be created by WIT-J.

<u>Post-Implosion Pegging Methods</u>

Class: **`Problem`**
Method: `void` **`clearPipSeq`** `()`

Clears the PIP shipment sequence.

Class: **`Demand`**
Method: `void` **`appendToPipSeq`** `(int theShipPer, double incShipVol)`

- Appends the triple (`the Demand, theShipPer, incShipVol`) to the PIP shipment sequence.

Class: **`Problem`**
Method: `ArrayList <PeggingTriple <Demand>>` **`getPipSeq`** `()`

Returns the PIP shipment sequence.

Class: **Problem**
Method: void **buildPip** ()

- Builds a post-implosion pegging of the current implosion solution.

Concurrent Pegging Method

Class: **Problem**
Method: void **clearPegging** ()

- Clears the concurrent pegging.

Pegging Retrieval Methods

The following nine methods return each of the demand peggings that WIT computes, represented as an `ArrayList <PeggingTriple <C>>`:

Class: **Demand**
Methods:

```
ArrayList <PeggingTriple <Part>>       getConsVolPip     (int shipPer)
ArrayList <PeggingTriple <BopEntry>>   getCoExecVolPip   (int shipPer)
ArrayList <PeggingTriple <Operation>>  getExecVolPip     (int shipPer)
ArrayList <PeggingTriple <Part>>       getProdVolPip     (int shipPer)
ArrayList <PeggingTriple <Part>>       getSideVolPip     (int shipPer)
ArrayList <PeggingTriple <Substitute>> getSubVolPip      (int shipPer)
ArrayList <PeggingTriple <Part>>       getSupplyVolPip   (int shipPer)
ArrayList <PeggingTriple <Operation>>  getExecVolPegging (int shipPer)
ArrayList <PeggingTriple <Substitute>> getSubVolPegging  (int shipPer)
```

- In each case, the pegging returned is for the pegging attribute indicated in the name of the method
- The version of pegging returned is indicated by the last portion of the method's name: “`Pip`” for post-implosion pegging and “`Pegging`” for concurrent pegging.
- The pegging returned is for the `Demand` for which the method was called and for the shipment period indicated by `shipPer`.

The following seven methods return each of the operation peggings that WIT computes, represented as an `ArrayList <PeggingTriple <C>>`:

Class: **Operation**
Methods:

```
ArrayList <PeggingTriple <Part>>       getConsVolPip     (int execPer)
ArrayList <PeggingTriple <BopEntry>>   getCoExecVolPip   (int execPer)
ArrayList <PeggingTriple <Operation>>  getExecVolPip     (int execPer)
ArrayList <PeggingTriple <Part>>       getProdVolPip     (int execPer)
ArrayList <PeggingTriple <Part>>       getSideVolPip     (int execPer)
ArrayList <PeggingTriple <Substitute>> getSubVolPip      (int execPer)
ArrayList <PeggingTriple <Part>>       getSupplyVolPip   (int execPer)
```

- In each case, the pegging returned is for the `Operation` for which the method was called and for the execution period indicated by `execPer`.

Code Example:

See Section 19 on page 41. (`PeggingTriple <Part>, buildPip, getSupplyVolPip`)

12. Quasi-Attribute Methods

The following methods set and retrieve the values of state variables that are not modeled as `Attributes`. They are called "quasi-attribute methods".

Class: Part

Method: `Part.Category` **`getCategory`** `()`

- Returns the `Part.Category` that represents the value of the partCategory WIT attribute for the WIT part represented by the `Part`.

See the `newInstance` method of class `Part` (page 6) for an explanation of the `Part.Category enum`.

The WIT attribute optInitMethod is a multiple-choice attribute that takes on one of four possible values. These values are represented in WIT-J by the `OptInitMethod enum`, which has the following four constants:

- `HEURISTIC`
- `ACCELERATED`
- `SCHEDULE`
- `CRASH`

Setting and retrieving the value of this WIT attribute is accomplished in WIT-J by the following two methods:

Class: **`Problem`**

Method: `void` **`setOptInitMethod`** `(OptInitMethod theMethod)`

- Sets the value of global `optInitMethod` WIT attribute for the underlying `WitRun` to the value represented by `theMethod`.

Method: `OptInitMethod` **`getOptInitMethod`** `()`

- Returns the `OptInitMethod` that represents the value of global `optInitMethod` WIT attribute for the underlying `WitRun`.

Class: **`Problem`**

Method: `void` **`setObjectiveList`** `(String... theObjNames)`

- Sets the value of the global `objectiveList` WIT attribute for the underlying `WitRun` to the sequence of objective names given in `theObjNames`.

Class: **`Problem`**

Method: `void` **`setObjectiveList`** `(ArrayList <String> theObjNameList)`

- Sets the value of the global `objectiveList` WIT attribute for the underlying `WitRun` to the sequence of objective names given in `theObjNameList`.

Class: **`Problem`**

Method: `ArrayList <String>` **`getObjectiveList`** `()`

Returns the value of the global `objectiveList` WIT attribute for the underlying `WitRun`.

13. Message Control Methods

The purpose of class `MessageMgr` ("Message Manager") is to provide control over WIT's messages. Each `Problem` has its own `MessageMgr`, which is constructed when the `Problem` is constructed. Access to a `Problem`'s `MessageMgr` is provided by the following method:

Class: **`Problem`**
Method: `MessageMgr` **`getMessageMgr`** `()`

- Returns the `MessageMgr` owned by the `Problem`.

The following are wrapper methods of class `MessageMgr`:

Class: **`MessageMgr`**
Method: `void` **`setMesgFileName`** `(String theName)`
Method: `String` **`getMesgFileName`** `()`
Method: `void` **`setMesgFileAccessMode`** `(String theMode)`
Method: `String` **`getMesgFileAccessMode`** `()`
Method: `void` **`setMesgTimesPrint`** `(int theMsgNo, int theCount)`
Method: `int` **`getMesgTimesPrint`** `(int theMsgNo)`

- Each of these methods wraps a WIT API function with the corresponding canonical name, e.g. `witSetMesgFileName`.
- In `setMesgTimesPrint`, if `theCount < 0`, the message will be printed an unlimited number of times.
- In `getMesgTimesPrint`, if the message is to be printed an unlimited number of times, `-1` is returned.

Code Example:

See Section 17 on page 41. (`getMessageMgr` and `setMesgFileName`)

To enable a group of messages to be specified in a single method call, WIT-J provides the `enum MessageGroup`. This `enum` has the following two constants:

- **`INFORMATIONAL`**: Represents the set of all informational messages in WIT.
- **`WARNING`**: Represents the set of all warning messages in WIT.

The following method uses `enum MessageGroup`:

Class: **`MessageMgr`**
Method: `void` **`setMesgTimesPrint`** `(MessageGroup theGroup, int theCount)`

- This is a wrapper method for `witSetMesgTimesPrint` when a message group is being specified.
- If `theCount < 0`, the messages will be printed an unlimited number of times.

Class `MessageMgr` also has a few non-wrapper methods, including this one:

Class: **`MessageMgr`**
Method: `void` **`flushFile`** `()`

- Flushes WIT's message file.

A `MessageMgr` has a boolean state variable called "quiet", which affects execution of some of the methods of the `MessageMgr`. If the quiet state is false, WIT will issue an informational message when any of these methods is called. If the quiet state is true, WIT will not issue an informational message when any of these methods is called. The `MessageMgr` quiet state variable applies to the following methods of class `MessageMgr`:

- `setMesgFileName       (String theName)`
- `setMesgFileAccessMode (String theMode)`
- `setMesgTimesPrint     (int          theMsgNo, int theCount)`
- `setMesgTimesPrint     (MessageGroup theGroup, int theCount)`

The default value of the `MessageMgr` quiet state variable is `true`.

Control over the `MessageMgr` quiet state variable is provided by the following two methods:

Class: **`MessageMgr`**
Method: `void` **`setQuiet`** `(boolean theBool)`
Method: `boolean` **`isQuiet`** `()`

- The `setQuiet` method sets the value of the quiet state variable of the `MessageMgr` to `theBool`.
- The `isQuiet` method returns the value of the quiet state variable of the `MessageMgr`.
- These methods do not cause WIT to issue any messages.

14. Memory Management

In a pure Java program, memory management is usually a fairly simple matter: You just stop referencing an object when you are done with it and garbage collection takes care of the rest. But a WIT-J application program is not a pure Java program. In many cases, when an instance of a public class from the Java portion of WIT-J is created, one a more instances of classes from the C++ portion are also created. Furthermore, when a WIT-J `Problem` is created, WIT-J creates a corresponding instance of the C++ class `WitRun`, the main class in WIT's API. Since Java's garbage-collection facility has no knowledge of the C++ objects in WIT-J, it cannot be expected to automatically invoke their C++ destructors and free up their memory when they are no longer needed. Instead, WIT-J has its own process for deleting the C++ objects that it has created and it provides a public interface that allows the application program to control this process.

WIT-J's memory management capability is largely implemented through one public class: `PreservedObject`. A `PreservedObject` is an object in WIT-J that is withheld from garbage collection until it is “deactivated”. `PreservedObject` is a superclass of the following classes in WIT-J:

- `Problem`
- `Component`
- `MessageMgr`

where, as you'll recall, many classes in WIT-J inherit from class `Component`.

At any point in time, a `PreservedObject` is considered to be in one of two states: “active” or “inactive”. Initially, just after construction, a `PreservedObject` is active. When a `PreservedObject` is active, the following conditions hold:

- It is considered to be responsible for one or more C++ objects.
- WIT-J maintains (perhaps indirectly) a static reference to it.

These two conditions prevent an active `PreservedObject` from being garbage-collected and prevent any C++ objects that it is responsible for from being memory-leaked.

Class `PreservedObject` has a package-internal method called “`deactivate`”. When `deactivate` is called, the following events occur:

- The `PreservedObject` becomes inactive and stays inactive permanently.
- The C++ objects that it is responsible for are deleted.
- All of WIT-J's internal references to the `PreservedObject` are removed.
- Various other clean-up tasks may be performed as well.

Thus, once a `PreservedObject` has been deactivated, it becomes eligible for garbage-collection as soon as the application program removes all of its references to it.

Class `PreservedObject` has two public methods:

Class: **`PreservedObject`**
Method: `boolean` **`isActive`** `()`
Method: `String` **`toString`** `()`

The method `isActive` returns `true`, iff the `PreservedObject` is currently active.
The method `toString` is just an override from class `Object`.

When a `PreservedObject` is inactive, most of its non-static public methods are no longer eligible to be invoked. The only exceptions to this are the `isActive` and `toString` methods and any methods inherited from superclasses. If any other non-static public WIT-J method is called on an inactive `PreservedObject`, an exception is thrown. Also, if an inactive `PreservedObject` is passed as an argument to a public WIT-J method, an exception is thrown.

As mentioned above, the `deactivate` method of class `PreservedObject` is package-internal. Thus the application program cannot call it directly; rather, it is invoked implicitly during the execution of various public methods of WIT-J. Currently, all of the public methods in WIT-J that cause deactivation are methods of class `Problem`. They are:

- `deactivate`
- `clearData`
- `copyData`
- `purgeData`

The methods `clearData`, `copyData`, and `purgeData` will be discussed in Section 15. The `deactivate` method will be explained here:

Class: **`Problem`**
Method: `void` **`deactivate`** `()`

This is class `Problem`'s public override of the package-internal `deactivate` method of class `PreservedObject`. This method:

- Deactivates the `Problem`.
- Deactivates all `Components` that belong to the `Problem`.
- Deactivates the `MessageMgr` that belongs to the `Problem`.
- Deletes all WIT-J/C++ objects for which the `Problem` is responsible.
- Deletes the underlying `WitRun` of the `Problem`.

Thus it is very important that the application program calls `deactivate` on any `Problem` that it no longer needs. Failure to do so would constitute a major memory leak.

<u>Code Example:</u>

See Section 17on page 41.

In summary, the following points should be kept in mind regarding memory management in WIT-J:

- Deactivate any `Problem` that has been created, once the application program no longer needs it.
- Be aware of the methods that cause deactivation.
- Don't use an inactive `PreservedObject` in any WIT-J method except `isActive` and `toString`.

15. Methods that Cause Deactivation

As stated in Section 14, several of WIT-J's methods cause deactivation of one or more PreservedObjects. This section describes these methods.

Class: **`Problem`**
Method: `void` **`deactivate`** `()`

This method was discussed in Section 14.

Class: **`Problem`**
Method: `void` **`clearData`** `()`

- This method invokes `witInitialize` on the underlying `WitRun`.
- All `Components` that belong to the `Problem` are deactivated, except the `Problem` itself. These `Components` no longer belong to the `Problem` and all internal references to them are removed.
- All global attributes are restored to their default values.
- The message subsystem is left unchanged.

Class: **`Problem`**
Method: `void` **`copyData`** `(Problem origProblem)`

- If the `Problem` and `origProblem` are the same object, no action is taken.
- Otherwise, this method starts by making an internal call to the `clearData` method.
- Then, `witCopyData` is invoked on the underlying `WitRun` of the `Problem` and the underlying `WitRun` of `origProblem`. This copies the WIT components in the underlying `WitRun` of `origProblem` (along with their input attribute values) into the underlying `WitRun` of the `Problem`.
- Finally, for each WIT component that has been copied into the underlying `WitRun`, a corresponding `Component` is constructed in WIT-J and associated with the `Problem`.

Class: **`Problem`**
Method: `void` **`purgeData`** `()`

- This method invokes `witPurgeData` on the underlying `WitRun`. This deletes WIT components from the `WitRun` based on the selForDel attribute.
- For each WIT component that has been deleted in the underlying `WitRun`, the corresponding WIT-J `Component` is deactivated, the `Component` no longer belongs to the `Problem`, and all internal references to it are removed.
- As explained in the WIT Guide, in addition to deleting the components for which the selForDel attribute is true, `witPurgeData` will also delete any component that has a prerequisite object that is being deleted. For example, if an operation is being deleted, then all BOM entries and BOP entries associated with it will also be deleted. When a WIT component is deleted by `witPurgeData` due to its prerequisites, the corresponding WIT-J `Component` will be deactivated.

16. Thread Safety

WIT-J's approach to thread safety is implemented by the use of a special class devoted to this purpose: `ThreadedObject`. A `ThreadedObject` is an object in WIT-J that is associated with a particular `Thread`: the `Thread` in which the object was created. Use of a `ThreadedObject` is limited to its associated `Thread`; use of a `ThreadedObject` from any other `Thread` triggers an exception. `ThreadedObject` is a direct superclass of class `PreservedObject`, and as you'll recall, many classes in WIT-J inherit from class `PreservedObject`. Note that `Attributes` are not `ThreadedObjects`. They don't need to be, because they are immutable.

Class `ThreadedObject` has the following method:

Class: **`ThreadedObject`**
Method: `Thread` **`getThread`** `()`

- Returns the `Thread` associated with the `ThreadedObject`.

WIT-J enforces the following thread safety rules:

- In general, if a non-static method of a `ThreadedObject` is called from outside its associated `Thread`, an exception is thrown. The only exceptions to this are the `getThread` method and any method inherited from class `Object`.
- If a `ThreadedObject` is passed as an argument to any WIT-J method from outside the `ThreadedObject`'s associated `Thread`, an exception is thrown.

These two rules have the following consequences:

- When a `ThreadedObject` is created by a factory method that requires another `ThreadedObject` as an argument, the newly created `ThreadedObject` will be associated with the same `Thread` as the argument `ThreadedObject`.
- All of the `ThreadedObject`s associated with a particular `Problem` will be associated with same `Thread` as the `Problem`. This includes all of the `Problem's Components` as well as its `MessageMgr`.
- The `origProblem` argument of the `copyData` method of class `Problem` will need to be associated with the same `Thread` as the `Problem` on which the method is called.
- The `origComp` argument of the `copyComponentData` method of class `Component` will need to be associated with the same `Thread` as the `Component` on which the method is called.

Note that many of the arguments to WIT-J methods are instances of standard Java classes and are therefore not `ThreadedObject`s. It is the responsibility of the application program to ensure that these argument objects are not updated concurrently while the WIT-J method is executing, since WIT-J does not check for this case.

As for synchronization, the `readData` method of class `Problem` is statically synchronized. This is done, because WIT does not permit asynchronous calls to `witReadData`. Other than `readData,` very little else in WIT-J is synchronized.

The general situation is this: A WIT-J application program is free to create multiple `Threads` and to create and populate multiple `Problems` in each `Thread`. The `Threads` may proceed asynchronously as usual and even interact with each other, as long as no `Thread` ever interacts with a `ThreadedObject` created in a different `Thread`.

17. Exceptions

When an error condition occurs during the execution of a WIT-J method, WIT-J throws an exception. When WIT-J throws an exception, the exception is always an instance of a class that's part of the WIT-J package. The following class hierarchy shows all of the exception classes that are thrown by WIT-J:

```
java.lang.RunTimeException
   WitjException
      StatusQuoException
      TerminalException
         TerminalAppException
         ReadDataException
         BadCharacterException
         OutOfMemoryException
         InternalErrorException
```

Note that, since class `java.lang.RunTimeException` is at the top of the hierarchy, it follows that all exceptions thrown by WIT-J are unchecked exceptions. Details on WIT-J's exception classes are as follows:

`WitjException:`

Abstract exception class.
Extends class `java.lang.RunTimeException`.
All exceptions thrown by WIT-J belong to subclasses of this class.

`StatusQuoException:`

Concrete exception class.
Extends class `WitjException`.
When WIT-J throws a `StatusQuoException`, this indicates that:

- An error condition was encountered that appears to be the result of an error in the application program.
- The state of WIT-J was not changed by the current invocation of a WIT-J public method.

Most error conditions detected by WIT-J itself generate `StatusQuoExceptions`. Specifically, anywhere in this document prior to this section, if it is stated that "an exception is thrown", the exception belongs to class `StatusQuoException`. The precise nature of the error condition is specified in the exception's detail message. When the application program catches such this kind of exception, it is free to make further calls to WIT-J, but one must be aware that a bug in the application program seems to have been found. A reasonable response to catching a `StatusQuoException` might be to display the detail message, deactivate the `Problem,` and gracefully shut down the application program.

`TerminalException:`

Abstract exception class.
Extends class `WitjException`.
When WIT-J throws a `TerminalException`, this indicates that an error condition was encountered that has put WIT-J into a "terminated" state: No further calls to WIT-J methods should be made for the rest of the program process, except for the following method, which is allowed:

Class: **`TerminalException`**
Method: `static boolean` **`witjHasTerminated`** `()`

- Returns `true`, iff WIT-J is currently in a terminated state.

If any other WIT-J method is called when WIT-J is in a terminated state, a `StatusQuoException` is thrown. A reasonable response to catching a `TerminalException` might be to display the detail message and gracefully shut down the application program without deactivating the `Problem`.

`TerminalAppException`:

Concrete exception class.
Extends class `TerminalException`.
When WIT-J throws a `TerminalAppException`, this indicates that:

- An error condition was encountered that appears to be the result of an error in the application program.
- As a result of the error condition, WIT-J has entered into a “terminated” state

In particular, WIT-J throws a `TerminalAppException` when WIT has issued a severe error message. In this case, information about the error is given in WIT's message file and in `stderr`.

`ReadDataException`:

Concrete exception class.
Extends class `TerminalException`.
When WIT-J throws a `ReadDataException`, this indicates that WIT has issued a severe error message while attempting to read a data file for the `readData` method of class `Problem`. Information about the error is given in WIT's message file and in `stderr`.

`BadCharacterException`:

Concrete exception class.
Extends class `TerminalException`.
When WIT-J throws a `BadCharacterException`, this indicates that the `readData` method of class `Problem` found a string in the data file containing a non-UTF-8 character. The exception's detail message shows the offending string, converted to a Java `String`. This class provides the following public method:

Class: **`BadCharacterException`**
Method: `static String` **`getOffendingString`** `()`

- Returns the string that contained the non-UTF-8 character, converted to a Java `String`.

`OutOfMemoryException`:

Concrete exception class.
Extends class `TerminalException`.
When WIT-J throws an `OutOfMemoryException`, this indicates that the program process has run out of memory. There are four distinct scenarios in which this class of exception can be thrown:

- The Java portion of WIT-J has run out of memory.
- The C++ portion of WIT-J has run out of memory.
- JNI has run out of memory.
- WIT has run out of memory.

The exception's detail message indicates which of the four scenarios has occurred. If it was the Java portion of WIT-J that ran out of memory, the original `java.lang.OutOfMemoryError` that was thrown by the JVM will be stored as the “cause” of the `OutOfMemoryException`.

`InternalErrorException`:

Concrete exception class.
Extends class `TerminalException`.
When WIT-J throws an `InternalErrorException`, this indicates that a bug in WIT-J has been detected. The exception's detail message provides information about the bug. In some cases, an exception of this class will be thrown in response to another exception that was thrown within the Java portion of WIT-J (e.g. a `java.lang.AssertionError`). In such a case, the original exception will be stored as the “cause” of the `InternalErrorException`.

18. Capabilities Not Implemented

The following WIT capabilities are not implemented in WIT-J and are not planned for any future release:

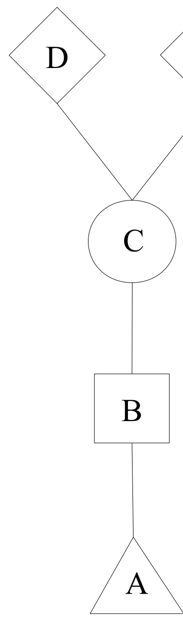
- Any WIT API function that whose argument list includes at least one argument of a type based on `float`. WIT-J only supports the double precision variants of these functions.
- `wit{Set/Get}{...}AppData` WIT-J does not provide this capability, because the Java generic class `HashMap` already provides it. Specifically, the WIT-J classes do not overload the `equals` or `hashCode` methods from class `Object`, so a `HashMap` that uses a WIT-J class as its key type will associate a value with each distinct instance of the class. Thus, e.g., a `HashMap <Part, V>` would be a completely effective way to associate a reference to an instance of type `V` with each `Part`.
- External Optimizing Implosion. This is best used in a C/C++ environment.
- `witFree` WIT-J invokes this as needed.
- `wit{Set/Get}MesgStopRunning` WIT-J needs this message attribute to be false at all times.
- `wit{Set/Get}MesgThrowErrorExc` WIT-J needs this message attribute to be true at all times.
- `wit{Set/Get}MesgPrintNumber`
- `wit{Set/Get}MesgFile`
- Reading an input data file into a non-empty Problem.
- The wit34Compatible attribute
- `witAddPartWithOperation`
- `witGetFocusShortageVolDbl` WIT-J supports focusShortageVol as a `Part Attribute` only.
- `witIncHeurAllocTwme` and `witEqHeurAllocTwme`
- Various undocumented capabilities included in WIT for upward compatibility, etc.

19. Example WIT-J Application Program

This section presents an example of a short, illustrative WIT-J application program.

Consider the (trivial) implosion problem shown in the following WIT diagram:

Figure 1: WIT Diagram for the Example WIT-J Application Program



This problem has the following attribute data:

```
Global:   nPeriods       = 2
Global:   pipSeqFromHeur = true
Part C:   buildNstn      = true
Part A:   supplyVol      = (30,  0)
Part C:   supplyVol      = (30, 10)
Demand D: demandVol      = (20, 20)
Demand E: demandVol      = (20, 20)
Demand D: priority       = 1
Demand E: priority       = 2
```

The next 5 pages show a WIT-J application program that performs the following actions:

- Constructs this problem.
- Calls heuristic implosion and PIP.
- Displays the shipment schedule and the supplyVol pegging from PIP.

This program illustrates the following aspects of WIT-J:

- Factory Methods
- Navigation Methods
- Setting Attribute Values
- Retrieving Attribute Values
- Wrapper Methods
- Pegging Methods
- Message Control Methods
- Memory Management
- Exceptions

Specific notes:

- The program uses method `setMesgFileName` of class `MessageMgr` to re-direct WIT's messages to the file `witj-demo1.wmsg`.
- The references returned by the factory methods for classes `Part`, `Operation`, and `Demand` are stored in local variables, so that these `Components` can be referred to later in method `buildProblem`.
- In contrast, the references returned by the factory methods for classes `BomEntry` and `BopEntry` are not stored, because the application program does not need to refer to these `Components.`
- The reference to the `Problem` is stored as a field in class `Demo1`.
- The `displaySolution` method uses navigation to iterate through all of the `Demands`.
- The `performDemo` method deactivates the `Problem` and removes its reference to it when it is finished with it.
- No exceptions are actually thrown by WIT-J during the execution of this program, but if an exception were thrown, it would be caught by the catch clause in the `performDemo` method and handled by the `handleWitjException` method. This method displays the exception and its stack trace, deactivates the `Problem` and removes its reference, if appropriate, and exits the program.

```
//==============================================================================
// Example WIT-J Application Program WitJDemo1.
//
// The program builds a trivial implosion problem, solves it with heuristic
// implosion and displays the shipment schedule and the supplyVol PIP.
//==============================================================================

import        com.ibm.witj.*;
import static com.ibm.witj.Attribute.*;
import static com.ibm.witj.Part.Category.*;

import                java.util.*;

public final class WitJDemo1
{
//------------------------------------------------------------------------------
// Main Program
//------------------------------------------------------------------------------

   public static void main (String[] theArgs)
      {
      WitJDemo1 theDemo;

      theDemo = new WitJDemo1 ();

      theDemo.performDemo ();
      }

//------------------------------------------------------------------------------
// myProblem
//
// The WIT-J Problem for this WIT-J application.
//------------------------------------------------------------------------------

   private Problem myProblem = null;

//------------------------------------------------------------------------------
// Constructor
//------------------------------------------------------------------------------

   private WitJDemo1 ()
      {
      }
```

```
//------------------------------------------------------------------------------
// performDemo ()
//------------------------------------------------------------------------------

   private void performDemo ()
      {
      try
         {
         buildProblem ();

         myProblem.heurImplode ();
         myProblem.buildPip    ();

         displaySolution ();

         myProblem.deactivate ();

         myProblem = null;
         }

      catch (WitjException theWitjException)
         {
         handleWitjException (theWitjException);
         }
      }
```

```
//------------------------------------------------------------------------------
// buildProblem ()
//
// Creates and populates myProblem.
//------------------------------------------------------------------------------

   private void buildProblem ()
      {
      Part      thePartA;
      Operation theOpnB;
      Part      thePartC;
      Demand    theDemandD;
      Demand    theDemandE;

      myProblem = Problem.newInstance ();

      myProblem.getMessageMgr ().setMesgFileName ("witj-demo1.wmsg");

      myProblem.set (N_PERIODS,         2);
      myProblem.set (PIP_SEQ_FROM_HEUR, true);

      thePartA   = Part     .newInstance (myProblem, "A",      CAPACITY);
      theOpnB    = Operation.newInstance (myProblem, "B");
                   BomEntry .newInstance (theOpnB,   thePartA);
      thePartC   = Part     .newInstance (myProblem, "C",      MATERIAL);
                   BopEntry .newInstance (theOpnB,   thePartC);
      theDemandD = Demand   .newInstance (thePartC,  "D");
      theDemandE = Demand   .newInstance (thePartC,  "E");

      thePartC  .set (BUILD_NSTN, true);

      thePartA  .set (SUPPLY_VOL, new double[]{30,  0});
      thePartC  .set (SUPPLY_VOL, new double[]{30, 10});

      theDemandD.set (DEMAND_VOL, new double[]{20, 20});
      theDemandE.set (DEMAND_VOL, new double[]{20, 20});

      theDemandD.setVectorToScalar (PRIORITY, 1);
      theDemandE.setVectorToScalar (PRIORITY, 2);
      }
```

```
//------------------------------------------------------------------------------
// displaySolution ()
//
// Displays the shipment schedule and the supplyVol pegging from PIP.
//------------------------------------------------------------------------------

   private void displaySolution ()
      {
      double[]                         shipVolArr;
      int                              thePer;
      ArrayList <PeggingTriple <Part>> supplyVolPip;

      for (Demand theDemand: myProblem.getDemands ())
         {
         shipVolArr = theDemand.get (SHIP_VOL);

         for (thePer = 0; thePer < 2; thePer ++)
            {
            if (shipVolArr[thePer] < .000001)
               continue;

            System.out.printf (
               "%n"
             + "Part %s, Demand %s, Period %d:          shipVol = %2.0f%n",
               theDemand.getPart ().get (PART_NAME),
               theDemand.get (DEMAND_NAME),
               thePer,
               shipVolArr[thePer]);

            supplyVolPip = theDemand.getSupplyVolPip (thePer);

            for (PeggingTriple <Part> theTriple: supplyVolPip)
               {
               System.out.printf (
                  "Part %s,           Period %d: pegged supplyVol = %2.0f%n",
                  theTriple.getRoot   ().get (PART_NAME),
                  theTriple.getPeriod (),
                  theTriple.getVolume ());
               }
            }
         }
      }
```

```
//----------------------------------------------------------------------------
// handleWitjException (theWitjException)
//
// Handles any exception thrown from WIT-J.
//----------------------------------------------------------------------------

   private void handleWitjException (WitjException theWitjException)
      {
      theWitjException.printStackTrace ();

      if (! TerminalException.witjHasTerminated ())
         {
         if (myProblem != null)
            {
            if (myProblem.isActive ())
               {
               myProblem.deactivate ();
               }

            myProblem = null;
            }
         }

      System.out.printf (
         "%n"
       + "WitJDemo1 is terminating due to the above exception.%n");

      System.exit (3);
      }
}
```

This program produces the following output:

```
Part C, Demand D, Period 0:          shipVol = 20
Part C,           Period 0: pegged supplyVol = 20

Part C, Demand D, Period 1:          shipVol = 20
Part C,           Period 1: pegged supplyVol = 10
Part A,           Period 0: pegged supplyVol = 10

Part C, Demand E, Period 0:          shipVol = 20
Part C,           Period 0: pegged supplyVol = 10
Part A,           Period 0: pegged supplyVol = 10

Part C, Demand E, Period 1:          shipVol = 10
Part A,           Period 0: pegged supplyVol = 10
```

20. Table of Attributes

The following table lists all of the `Attributes` in WIT-J, along with their most important properties:

Table 1: Attributes

Attribute	Value Type	Applies To	Modifiable?
ACCELERATED	Boolean	Problem	false
ACC_AFTER_OPT_IMP	Boolean	Problem	false
ACC_AFTER_SOFT_L_B	Boolean	Problem	false
ASAP_PIP_ORDER	Boolean	Part	true
AUTO_PRIORITY	Boolean	Problem	true
BOM_INDEX	Integer	BomEntry	false
BOP_INDEX	Integer	BopEntry	false
BOUNDED_LEAD_TIMES	Boolean	Part	true
BOUNDS_VALUE	Double	Problem	false
BUILD_AHEAD_U_B	int[]	Part	true
BUILD_ASAP	Boolean	Part	true
BUILD_NSTN	Boolean	Part	true
COMPUTE_CRITICAL_LIST	Boolean	Problem	true
COMP_PRICES	Boolean	Problem	false
CONS_RATE	double[]	BomEntry Substitute	true
CONS_VOL	double[]	Part	false
CPLEX_EMBEDDED	Boolean	Problem	false
CPLEX_MIP_BOUND	Double	Problem	false
CPLEX_MIP_REL_GAP	Double	Problem	false
CPLEX_PAR_SPEC_DBL_VAL	Double	Problem	true
CPLEX_PAR_SPEC_INT_VAL	Integer	Problem	true
CPLEX_PAR_SPEC_NAME	String	Problem	true
CPLEX_STATUS_CODE	Integer	Problem	false
CPLEX_STATUS_TEXT	String	Problem	false
CUM_SHIP_BOUNDS	BoundSet	Demand	true
CUM_SHIP_REWARD	double[]	Demand	true
CURRENT_OBJECTIVE	String	Problem	true
CURRENT_SCENARIO	Integer	Problem	false
DEMAND_NAME	String	Demand	true
DEMAND_VOL	double[]	Demand	true
EARLIEST_PERIOD	Integer	BomEntry Substitute BopEntry	true
EQUITABILITY	Integer	Problem	true

Attribute	Value Type	Applies To	Modifiable?
EXCESS_VOL	double[]	Part	false
EXECUTABLE	boolean[]	Operation	false
EXEC_BOUNDS	BoundSet	Operation	true
EXEC_COST	double[]	Operation	true
EXEC_EMPTY_BOM	Boolean	Problem	true
EXEC_PENALTY	Double	Operation BomEntry Substitute	true
EXEC_VOL	double[]	Operation	true
EXP_ALLOWED	Boolean	Substitute BopEntry	true
EXP_AVERSION	Double	BopEntry	true
EXP_CUTOFF	Double	Problem	true
EXP_NET_AVERSION	Double	Substitute	true
EXT_OPT_ACTIVE	Boolean	Problem	false
FALLOUT_RATE	Double	BomEntry Substitute	true
FEASIBLE	Boolean	Problem	false
FOCUS_HORIZON	Integer	Demand	true
FOCUS_SHORTAGE_VOL	double[]	Part	false
FORCED_MULTI_EQ	Boolean	Boolean	true
FSS_EXEC_VOL	double[]	Operation	false
FSS_SHIP_VOL	double[]	Demand	true
FSS_SUB_VOL	double[]	Substitute	false
HEUR_ALLOC_ACTIVE	Boolean	Problem	false
HIGH_PRECISION_W_D	Boolean	Problem	true
IMPACT_PERIOD	int[]	BomEntry Substitute BopEntry	false
INC_LOT_SIZE	double[]	Operation	true
INC_LOT_SIZE2	double[]	Operation	true
INDEPENDENT_OFFSETS	Boolean	Problem	true
INT_EXEC_VOLS	Boolean	Operation	true
INT_SHIP_VOLS	Boolean	Demand	true
INT_SUB_VOLS	Boolean	Substitute	true
LATEST_PERIOD	Integer	BomEntry Substitute BopEntry	true
LEAD_TIME_U_B	int[]	Demand	true
LOT_SIZE2_THRESH	double[]	Operation	true
LOT_SIZE_TOL	Double	Problem	true

Attribute	Value Type	Applies To	Modifiable?
MAND_E_C	Boolean	BomEntry	true
MINIMAL_EXCESS	Boolean	Problem	false
MIN_LOT_SIZE	double[]	Operation	true
MIN_LOT_SIZE2	double[]	Operation	true
MIP_MODE	Boolean	Problem	true
MOD_HEUR_ALLOC	Boolean	Problem	true
MRP_CONS_VOL	double[]	Part	false
MRP_EXCESS_VOL	double[]	Part	false
MRP_EXEC_VOL	double[]	Operation	false
MRP_NET_ALLOWED	Boolean	Substitute	true
MRP_RESIDUAL_VOL	double[]	Part	false
MRP_SUB_VOL	double[]	Substitute	false
MULTI_EXEC	Boolean	Problem	true
MULTI_OBJ_MODE	Boolean	Problem	true
MULTI_OBJ_TOL	Double	Problem	true
MULTI_ROUTE	Boolean	Problem	true
NET_ALLOWED	Boolean	Substitute	true
NSTN_RESIDUAL	Boolean	Problem	true
N_PERIODS	Integer	Problem	false
N_SCENARIOS	Integer	Problem	false
OBJECTIVE_LIST_SPEC	String	Problem	true
OBJECTIVE_SEQ_NO	Integer	Problem	true
OBJECT_STAGE	Integer	Part Operation	false
OBJ_VALUE	Double	Problem	false
OFFSET	double[]	BomEntry Substitute BopEntry	true
OPERATION_NAME	String	Operation	true
PART_NAME	String	Part	true
PEN_EXEC	Boolean	Problem	true
PERF_PEGGING	Boolean	Problem	true
PERIOD_STAGE	int[]	Problem	true
PGD_CRIT_LIST_MODE	Boolean	Problem	true
PIP_ENABLED	Boolean	Operation	true
PIP_EXISTS	Boolean	Problem	false
PIP_RANK	Integer	Operation	true
PIP_SEQ_FROM_HEUR	Boolean	Problem	true
POSTPROCESSED	Boolean	Problem	false
PREF_HIGH_STOCK_S_L_BS	Boolean	Problem	true

Attribute	Value Type	Applies To	Modifiable?
PREPROCESSED	Boolean	Problem	false
PRIORITY	int[]	Demand	true
PROBABILITY	Double	Problem	true
PRODUCT_RATE	double[]	BopEntry	true
PROD_VOL	double[]	Part	false
PROP_RTG	boolean[]	Part BomEntry	true
REQ_VOL	double[]	Part	false
RESIDUAL_VOL	double[]	Part	false
RESPECT_STOCK_S_L_BS	Boolean	Problem	true
ROUND_REQ_VOLS	Boolean	Problem	true
ROUTING_SHARE	double[]	BomEntry Substitute BopEntry	true
SCRAP_ALLOWED	Boolean	Part	true
SCRAP_COST	double[]	Part	true
SCRAP_VOL	double[]	Part	false
SEARCH_INC	Double	Demand	true
SELECTION_RECOVERY	Boolean	Problem	false
SEL_FOR_DEL	Boolean	Part Demand Operation BomEntry Substitute BopEntry	true
SEL_SPLIT	Boolean	Problem	true
SHADOW_PRICE	double[]	Part	false
SHIP_LATE_ALLOWED	Boolean	Demand	true
SHIP_LATE_U_B	int[]	Demand	true
SHIP_REWARD	double[]	Demand	true
SHIP_VOL	double[]	Demand	true
SINGLE_SOURCE	Boolean	Part BomEntry	true
SKIP_FAILURES	Boolean	Problem	true
SOLVER_LOG_FILE_NAME	String	Problem	true
STAGE_BY_OBJECT	Boolean	Problem	true
STOCH_MODE	Boolean	Problem	true
STOCH_SOLN_MODE	Boolean	Problem	false
STOCK_BOUNDS	BoundSet	Part	true
STOCK_COST	double[]	Part	true
STOCK_REALLOCATION	Boolean	Problem	true
STOCK_VOL	double[]	Part	false

Attribute	Value Type	Applies To	Modifiable?
SUB_COST	double[]	Substitute	true
SUB_INDEX	Integer	Substitute	false
SUB_VOL	double[]	Substitute	true
SUPPLY_VOL	double[]	Part	true
TIE_BREAK_PROP_RT	Boolean	Problem	true
TITLE	String	Problem	true
TRUNC_OFFSETS	Boolean	Problem	true
TWO_LEVEL_LOT_SIZES	Boolean	Operation	true
TWO_WAY_MULTI_EXEC	Boolean	Problem	true
USER_HEUR_START	Boolean	Problem	true
USE_FOCUS_HORIZONS	Boolean	Problem	false
WBOUNDS	Double	Problem	true
YIELD_RATE	double[]	Operation	true

21. Class Listing

This section consists of an alphabetical annotated list of all public classes and enums in WIT-J. For each public class/enum, the following information is provided:

- The location of the class/enum in the class hierarchy
- Cross references to where detailed information on the class/enum can be found
- An alphabetical list of the public methods of the class/enum, if it has any
- A cross reference to where each public method is described
- In the case of an enum, a list of its constants

Specifically, each cross reference gives the section number and starting page number of the relevant section.

Class: **`Attribute <V>`**

```
java.lang.Object
   Attribute <V>
```

Main Description:

- Section 6 (page 12): Introduction to `Attributes`
- Section 7 (page 14): Methods for Setting `Attribute` Values
- Section 8 (page 16): Methods for Retrieving `Attribute` Values
- Section 9 (page 18): Methods for Working with `Attributes`.
- Section 20 (page 48): Table of `Attributes`

Public Methods:

```
appliesTo
asAttOfType
getAttributes
getValueType
hasDefaultValue
getDefaultValue
getDefaultBoundSet
isModifiable
isValidFor
toString
```

Each of these methods is described in Section 9 (page 18).

Public Fields:

As explained in Section 6, generic class `Attribute <V>` has one public static final field for each instance. For example:

Field: `static final Attribute <double[]>` **`SUPPLY_VOL`**

There are currently 148 `Attributes` in WIT-J. They are listed in Table 1 on page 48.

Class: **`BadCharacterException`**

```
java.lang.RunTimeException
   WitjException
      TerminalException
         BadCharacterException
```

Main Description:

- Section 17 (page 38): Exceptions

Public Methods:

`getOffendingString`	Section 17 (page 38)

Class: **`BomEntry`**

```
java.lang.Object
   ThreadedObject
      PreservedObject
         Component
            BomEntry
```

Brief Description:

- Section 3 (page 5): Class Overview

Also Appears In:

- Section 4 (page 6): Factory Methods
- Section 5 (page 9): Navigation Methods
- Section 20 (page 48): Table of `Attributes`

Public Methods:

`getOperation`	Section 5 (page 9)
`getPart`	Section 5 (page 9)
`getSubstitutes`	Section 5 (page 9)
`getSubstitutesTo`	Section 5 (page 9)
`getUniqueSubstituteTo`	Section 5 (page 9)
`newInstance`	Section 4 (page 6)

Class: **`BopEntry`**

```
java.lang.Object
   ThreadedObject
      PreservedObject
         Component
            BopEntry
```

Brief Description:

- Section 3 (page 5): Class Overview

Also Appears In:

- Section 4 (page 6): Factory Methods
- Section 5 (page 9): Navigation Methods
- Section 11 (page 28): Pegging Methods
- Section 20 (page 48): Table of `Attributes`

Public Methods:

`getOperation`	Section 5 (page 9)
`getPart`	Section 5 (page 9)
`newInstance`	Section 4 (page 6)

Class: **`BoundSet`**

```
java.lang.Object
   BoundSet
```

Main Description:

- Section 6 (page 12): Introduction to `Attributes`

Also Appears In:

- Section 7 (page 14): Methods for setting `Attribute` values
- Section 8 (page 16): Methods for retrieving `Attribute` values
- Section 9 (page 18): Methods for working with `Attributes`.
- Section 20 (page 48): Table of `Attributes`

Class: **`Component`**

```
java.lang.Object
   ThreadedObject
      PreservedObject
         Component
            BomEntry
            BopEntry
            Demand
            Operation
            Part
            Problem
            Substitute
```

Brief Description:

- Section 3 (page 5): Class Overview

Also Appears In:

- Many other sections

Public Methods:

(Some of these method names are overloaded.)

`copyComponentData`	Section 10 (page 22)
`get`	Section 8 (page 16)
`getBoundSet`	Section 8 (page 16)
`getVector`	Section 8 (page 16)
`getProblem`	Section 5 (page 9)
`set`	Section 7 (page 14)
`setBoundSet`	Section 7 (page 14)
`setBoundSetToScalars`	Section 7 (page 14)
`setVectorToScalar`	Section 7 (page 14)

Class: **`Demand`**

```
java.lang.Object
   ThreadedObject
      PreservedObject
         Component
            Demand
```

Brief Description:

- Section 3 (page 5): Class Overview

Also Appears In:

- Section 4 (page 6): Factory Methods
- Section 5 (page 9): Navigation Methods
- Section 10 (page 22): Wrapper Methods
- Section 11 (page 28): Pegging Methods

Public Methods:

`getCoExecVolPip`	Section 11 (page 28)
`getConsVolPip`	Section 11 (page 28)
`getExecVolPegging`	Section 11 (page 28)
`getExecVolPip`	Section 11 (page 28)
`getPart`	Section 5 (page 9)
`getProdVolPip`	Section 11 (page 28)
`getSideVolPip`	Section 11 (page 28)
`getSubVolPegging`	Section 11 (page 28)
`getSubVolPip`	Section 11 (page 28)
`getSupplyVolPip`	Section 11 (page 28)
`incHeurAlloc`	Section 10 (page 22)
`newInstance`	Section 4 (page 6)

Enum: **`FileFormat`**

```
java.lang.Enum
   FileFormat
```

Main Description:

- Section 10 (page 22): Wrapper Methods

Constants:

- `BSV`
- `CSV`

Class: **InternalErrorException**

```
java.lang.RunTimeException
   WitjException
      TerminalException
         InternalErrorException
```

Main Description:

- Section 17 (page 38): Exceptions

Enum: **MessageGroup**

```
java.lang.Enum
   MessageGroup
```

Main Description:

- Section 13 (page 32): Message Control Methods

Constants:

- `INFORMATIONAL`
- `WARNING`

Class: **`MessageMgr`**

```
java.lang.Object
   ThreadedObject
      PreservedObject
         MessageMgr
```

Main Description:

- Section 3 (page 5): Class Overview
- Section 13 (page 32): Message Control Methods

Also Appears In:

- Section 4 (page 6): Factory Methods
- Section 14 (page 34): Memory Management

Public Methods:

`flushFile`
`getMesgFileAccessMode`
`getMesgFileName`
`getMesgTimesPrint`
`isQuiet`
`setMesgFileAccessMode`
`setMesgFileName`
`setMesgTimesPrint` (Overloaded)
`setQuiet`

Each of these methods is described in Section 13 (page 32).

Class: **Operation**

```
java.lang.Object
   ThreadedObject
      PreservedObject
         Component
            Operation
```

Brief Description:

- Section 3 (page 5): Class Overview

Also Appears In:

- Section 4 (page 6): Factory Methods
- Section 5 (page 9): Navigation Methods
- Section 11 (page 28): Pegging Methods
- Section 20 (page 48): Table of `Attributes`

Public Methods:

`getBomEntries`	Section 5 (page 9)
`getBomEntriesTo`	Section 5 (page 9)
`getBopEntries`	Section 5 (page 9)
`getBopEntriesTo`	Section 5 (page 9)
`getCoExecVolPip`	Section 11 (page 28)
`getConsVolPip`	Section 11 (page 28)
`getExecVolPip`	Section 11 (page 28)
`getProdVolPip`	Section 11 (page 28)
`getSideVolPip`	Section 11 (page 28)
`getSubVolPip`	Section 11 (page 28)
`getSupplyVolPip`	Section 11 (page 28)
`getUniqueBomEntryTo`	Section 5 (page 9)
`getUniqueBopEntryTo`	Section 5 (page 9)
`newInstance`	Section 4 (page 6)

Enum: **OptInitMethod**

```
java.lang.Enum
   OptInitMethod
```

Main Description:

- Section 12 (page 30): Quasi-Attribute Methods

Constants:

- `HEURISTIC`
- `ACCELERATED`
- `SCHEDULE`
- `CRASH`

Class: **`OutOfMemoryException`**

```
java.lang.RunTimeException
   WitjException
      TerminalException
         OutOfMemoryException
```

Main Description:

- Section 17 (page 38): Exceptions

Class: **`Part`**

```
java.lang.Object
   ThreadedObject
      PreservedObject
         Component
            Part
```

Brief Description:

- Section 3 (page 5): Class Overview

Also Appears In:

- Section 4 (page 6): Factory Methods
- Section 5 (page 9): Navigation Methods
- Section 11 (page 28): Pegging Methods
- Section 12 (page 30): Quasi-Attribute Methods
- Section 20 (page 48): Table of `Attributes`

Public Methods:

`getBomEntries`	Section 5 (page 9)
`getBopEntries`	Section 5 (page 9)
`getCategory`	Section 12 (page 30)
`getDemand`	Section 5 (page 9)
`getDemands`	Section 5 (page 9)
`getSubstitutes`	Section 5 (page 9)
`newInstance`	Section 4 (page 6)

Enum: **`Part.Category`**

```
java.lang.Enum
   Part.Category
```

Main Description:

- Section 4 (page 6): Factory Methods

Also Appears In:

- Section 12 (page 30): Quasi-Attribute Methods

Constants:

- `MATERIAL`
- `CAPACITY`

Class: **`PeggingTriple <C extends Component>`**

```
java.lang.Object
   PeggingTriple <C extends Component>
```

Main Description:

- Section 11 (page 28): Pegging Methods

Public Methods:

```
getPeriod
getRoot
getVolume
toString
```

Each of these methods is described in Section 11 (page 28).

Class: **PreservedObject**

```
java.lang.Object
   PreservedObject
      Component
         BomEntry
         BopEntry
         Demand
         Operation
         Problem
         Part
         Substitute
      MessageMgr
```

Main Description:

- Section 14 (page 34): Memory Management

Public Methods:

`isActive`	Section 14 (page 34)
`toString`	Section 14 (page 34)

Class: **Problem**

```
java.lang.Object
   ThreadedObject
      PreservedObject
         Component
            Problem
```

Brief Description:

- Section 3 (page 5): Class Overview

Also Appears In:

- Many other sections

Public Methods:

`addDblCplexParSpec`	Section 10 (page 22)
`addIntCplexParSpec`	Section 10 (page 22)
`appendToPipSeq`	Section 11 (page 28)
`buildPip`	Section 11 (page 28)
`clearCplexParSpecs`	Section 10 (page 22)
`clearData`	Section 15 (page 36)
`clearPegging`	Section 11 (page 28)
`clearPipSeq`	Section 11 (page 28)
`clearStochSoln`	Section 15 (page 22)
`copyData`	Section 15 (page 36)
`deactivate`	Section 14 (page 34)
`displayData`	Section 10 (page 22)

Class: **`ReadDataException`**

```
java.lang.RunTimeException
   WitjException
      TerminalException
         ReadDataException
```

Main Description:

- Section 17 (page 38): Exceptions

Class: **`StatusQuoException`**

```
java.lang.RunTimeException
   WitjException
      StatusQuoException
```

Main Description:

- Section 17 (page 38): Exceptions

Class: **`Substitute`**

```
java.lang.Object
   ThreadedObject
      PreservedObject
         Component
            Substitute
```

Brief Description:

- Section 3 (page 5): Class Overview

Also Appears In:

- Section 4 (page 6): Factory Methods
- Section 5 (page 9): Navigation Methods
- Section 11 (page 28): Pegging Methods
- Section 20 (page 48): Table of `Attributes`

Public Methods:

`getBomEntry`	Section 5 (page 9)
`getPart`	Section 5 (page 9)
`newInstance`	Section 4 (page 6)

Class: **`TerminalException`**

```
java.lang.RunTimeException
   WitjException
      TerminalException
         TerminalAppException
         ReadDataException
         BadCharacterException
         OutOfMemoryException
         InternalErrorException
```

Main Description:

- Section 17 (page 38): Exceptions

Public Methods:

`witjHasTerminated` Section 17 (page 38)

Class: **`TerminalAppException`**

```
java.lang.RunTimeException
   WitjException
      TerminalException
         TerminalAppException
```

Main Description:

- Section 17 (page 38): Exceptions

Class: **ThreadedObject**

```
java.lang.Object
   ThreadedObject
      PreservedObject
         Component
            BomEntry
            BopEntry
            Demand
            Operation
            Part
            Problem
            Substitute
         MessageMgr
```

Main Description:

- Section 16 (page 37): Thread Safety

Public Method:

`getThread` Section 16 (page 37)

Class: **WitjException**

```
java.lang.RunTimeException
   WitjException
      StatusQuoException
      TerminalException
         TerminalAppException
         ReadDataException
         BadCharacterException
         OutOfMemoryException
         InternalErrorException
```

Main Description:

- Section 17 (page 38): Exceptions

22. WIT API Functions and Corresponding WIT-J Capabilities

The following table lists each documented WIT API function and indicates the corresponding equivalent WIT-J capability, if any, specifying the WIT-J class, the WIT-J method, the section in which the WIT-J method is discussed, and the section number and starting page number of the relevant section. Note that the WIT-J method corresponding to a given WIT API function does not necessarily call the WIT API function; it merely provides the equivalent capability.

Table 2: WIT API Functions and Corresponding WIT-J Capabilities

WIT API Function(s)	WIT-J Class	WIT-J Method	Section	Sect # (Pg #)
`witAdd{Component Class}`	Various	`newInstance`	Factory Methods	4 (6)
`witAddDblCplexParSpec` `witAddIntCplexParSpec`	`Problem`	`addDblCplexParSpec` `addIntCplexParSpec`	Wrapper Methods	10 (22)
`witAddPartWithOperation`			Capabilities Not Implemented	18 (40)
`witAdvanceObjItr`	`Problem`	`getComponents`, `getParts`, etc.	Navigation Methods	5 (9)

`witAppendToPipSeq{Dbl}`
`witBuildPip`
`witClearPegging`
`witClearPip`
	Various	Various	Pegging Methods	11 (28)

`witClearCplexParSpecs`	`Problem`	`clearCplexParSpecs`	Wrapper Methods	10 (22)
`witClearStochSoln`	`Problem`	`clearStochSoln`	Wrapper Methods	10 (22)

`witCopyBomEntryData`
`witCopyBopEntryData`
`witCopyDemandData`
`witCopyOperationData`
`witCopyPartData`
`witCopySubsBomEntryData`
	`Component`	`copyComponentData`	Wrapper Methods	10 (22)

`witCopyData`	`Problem`	`copyData`	Methods that Cause Deactivation	15 (36)
`witDeleteRun`	`Problem`	`deactivate`	Memory Management	14 (34)
`witDisplayData`	`Problem`	`displayData`	Wrapper Methods	10 (22)
`witEqHeurAlloc{Dbl}`	`Problem`	`eqHeurAlloc`	Wrapper Methods	10 (22)
`witEqHeurAllocTwme{Dbl}`			Capabilities Not Implemented	18 (40)
`witEvalObjectives`	`Problem`	`evalObjectives`	Wrapper Methods	10 (22)
`witFinishHeurAlloc`	`Problem`	`finishHeurAlloc`	Wrapper Methods	10 (22)
`witFree`			Capabilities Not Implemented	18 (40)
`witGeneratePriorities`	`Problem`	`generatePriorities`	Wrapper Methods	10 (22)
`witGet{Attribute}`	`Component`	`get` [overloaded]	Retrieving `Attribute` Values	8 (16)
`witGetAppData`		Use `HashMap <Problem, V>`	Capabilities Not Implemented	18 (40)

WIT API Function(s)	WIT-J Class	WIT-J Method	Section	Sect # (Pg #)
witGetBomEntryAppData		Use HashMap <BomEntry, V>	Capabilities Not Implemented	18 (40)
witGetBomEntry{Attribute}	Component	get [overloaded]	Retrieving Attribute Values	8 (16)
witGetBomEntryConsumedPart	BomEntry	getPart	Navigation Methods	5 (9)
witGetBomEntryNSubsBomEntries	BomEntry	getSubstitutes	Navigation Methods	5 (9)
witGetBomEntryNonSubVarIndex witGetBomEntrySubConIndex			Capabilities Not Implemented	18 (40)
witGetBopEntryAppData		Use HashMap <BopEntry, V>	Capabilities Not Implemented	18 (40)
witGetBopEntry{Attribute}	Component	get [overloaded]	Retrieving Attribute Values	8 (16)
witGetBopEntryProducedPart	BopEntry	getPart	Navigation Methods	5 (9)
witGetCriticalList	Problem	getCriticalList	Wrapper Methods	10 (22)
witGetDblCplexParSpec witGetIntCplexParSpec	Problem	getDblCplexParSpec getIntCplexParSpec	Wrapper Methods	10 (22)
witGetDemandAppData		Use HashMap <Demand, V>	Capabilities Not Implemented	18 (40)
witGetDemandCoExecVolPip{Dbl}	Demand	getCoExecVolPip	Pegging Methods	11 (28)
witGetDemandConsVolPip{Dbl}	Demand	getConsVolPip	Pegging Methods	11 (28)
witGetDemand{Attribute}	Component	get [overloaded]	Retrieving Attribute Values	8 (16)
witGetDemandCumShipSlbConIndex witGetDemandCumShipSlbvVarIndex witGetDemandCumShipVarIndex			Capabilities Not Implemented	18 (40)
witGetDemandExecVolPegging{Dbl}	Demand	getExecVolPegging	Pegging Methods	11 (28)
witGetDemandExecVolPip{Dbl}	Demand	getExecVolPip	Pegging Methods	11 (28)
witGetDemandProdVolPip{Dbl}	Demand	getProdVolPip	Pegging Methods	11 (28)
witGetDemandShipConIndex witGetDemandShipVarIndex			Capabilities Not Implemented	18 (40)
witGetDemandSideVolPip{Dbl}	Demand	getSideVolPip	Pegging Methods	11 (28)
witGetDemandSubVolPegging{Dbl}	Demand	getSubVolPegging	Pegging Methods	11 (28)
witGetDemandSubVolPip{Dbl}	Demand	getSubVolPip	Pegging Methods	11 (28)
witGetDemandSupplyVolPip{Dbl}	Demand	getSupplyVolPip	Pegging Methods	11 (28)
witGetExpCycle	Problem	getExpCycle	Wrapper Methods	10 (22)
witGetExtOptIntVarIndices witGetExtOptLpProb{Dbl} witGetFocusShortageVol{Dbl} witGetMesgFile			Capabilities Not Implemented	18 (40)
witGetMesgFileAccessMode	MessageMgr	getMesgFileAccessMode	Message Control Methods	13 (32)
witGetMesgFileName	MessageMgr	getMesgFileName	Message Control	13 (32)

WIT API Function(s)	WIT-J Class	WIT-J Method	Section	Sect # (Pg #)
			Methods	
witGetMesgPrintNumber witGetMesgStopRunning witGetMesgThrowErrorExc			Capabilities Not Implemented	18 (40)
witGetMesgTimesPrint	MessageMgr	getMesgTimesPrint	Message Control Methods	13 (32)
witGetObjItr{Component Class} witGetObjItrState	Problem	getComponents, getParts, etc.	Navigation Methods	5 (9)
witGetObjectiveList	Problem	getObjectiveList	Quasi-Attribute Methods	12 (30)
witGetOperationAppData		Use HashMap <Operation, V>	Capabilities Not Implemented	18 (40)
witGetOperation{Attribute}	Component	get [overloaded]	Retrieving Attribute Values	8 (16)
witGetOperationCoExecVolPip{Dbl}	Operation	getCoExecVolPip	Pegging Methods	11 (28)
witGetOperationConsVolPip{Dbl}	Operation	getConsVolPip	Pegging Methods	11 (28)
witGetOperationExecSlbConIndex witGetOperationExecSlbvVarIndex witGetOperationExecVarIndex			Capabilities Not Implemented	18 (40)
witGetOperationExecVolPip{Dbl}	Operation	getExecVolPip	Pegging Methods	11 (28)
witGetOperationExists	Problem	getOperation	Navigation Methods	5 (9)
witGetOperationNBomEntries	Operation	getBomEntries	Navigation Methods	5 (9)
witGetOperationNBopEntries	Operation	getBopEntries	Navigation Methods	5 (9)
witGetOperationProdVolPip{Dbl}	Operation	getProdVolPip	Pegging Methods	11 (28)
witGetOperationSideVolPip{Dbl}	Operation	getSideVolPip	Pegging Methods	11 (28)
witGetOperationSubVolPip{Dbl}	Operation	getSubVolPip	Pegging Methods	11 (28)
witGetOperationSupplyVolPip{Dbl}	Operation	getSupplyVolPip	Pegging Methods	11 (28)
witGetOperations	Problem	getSortedOperations	Wrapper Methods	10 (22)
witGetOptInitMethod	Problem	getOptInitMethod	Quasi-Attribute Methods	12 (30)
witGetPartAppData		Use HashMap <Part, V>	Capabilities Not Implemented	18 (40)
witGetPart{Attribute}	Component	get [overloaded]	Retrieving Attribute Values	8 (16)
witGetPartBelowList	Part	getBelowList	Wrapper Methods	10 (22)
witGetPartCategory	Part	getCategory	Quasi-Attribute Methods	12 (30)
witGetPartConsumingBomEntry	Part	getBomEntries	Navigation Methods	5 (9)
witGetPartConsumingSubsBomEntry	Part	getSubstitutes	Navigation Methods	5 (9)
witGetPartDemands	Part	getDemands	Navigation Methods	5 (9)
witGetPartExists	Problem	getPart	Navigation Methods	5 (9)
witGetPartNConsumingBomEntries	Part	getBomEntries	Navigation Methods	5 (9)
witGetPartNConsumingSubsBomEntries	Part	getSubstitutes	Navigation Methods	5 (9)

WIT API Function(s)	WIT-J Class	WIT-J Method	Section	Sect # (Pg #)
witGetPartNProducingBomEntries	Part	getBopEntries	Navigation Methods	5 (9)
witGetPartProducingBomEntry	Part	getBopEntries	Navigation Methods	5 (9)
witGetPartResourceConIndex witGetPartScrapVarIndex witGetPartStockSlbConIndex witGetPartStockSlbvVarIndex witGetPartStockVarIndex			Capabilities Not Implemented	18 (40)
witGetParts	Problem	getSortedParts	Wrapper Methods	10 (22)
witGetPgdCritList	Problem	getPgdCritList	Wrapper Methods	10 (22)
witGetPipSeq{Dbl}	Problem	getPipSeq	Pegging Methods	11 (28)
witGetSubsBomEntryAppData		Use HashMap <Substitute, V>	Capabilities Not Implemented	18 (40)
witGetSubsBomEntry{Attribute}	Component	get [overloaded]	Retrieving Attribute Values	8 (16)
witGetSubsBomEntrySubVarIndex witGetWit34Compatible			Capabilities Not Implemented	18 (40)
witHeurImplode	Problem	heurImplode	Wrapper Methods	10 (22)
witIncHeurAlloc{Dbl}	Demand	incHeurAlloc	Wrapper Methods	10 (22)
witIncHeurAllocTwme{Dbl}			Capabilities Not Implemented	18 (40)
witInitialize	Problem	clearData	Methods that Cause Deactivation	15 (36)
	Problem	newInstance	Factory Methods	4 (6)
witMrp	Problem	mrp	Wrapper Methods	10 (22)
witNewRun	Problem	newInstance	Factory Methods	4 (6)
witOptImplode	Problem	optImplode	Wrapper Methods	10 (22)
witPostprocess	Problem	postprocess	Wrapper Methods	10 (22)
witPreprocess	Problem	preprocess	Wrapper Methods	10 (22)
witPurgeData	Problem	purgeData	Methods that Cause Deactivation	15 (36)
witReadData	Problem	readData	Wrapper Methods	10 (22)
witResetObjItr	Problem	getComponents, getParts, etc.	Navigation Methods	5 (9)
witSet{Attribute}	Component	set [overloaded]	Setting Attribute Values	7 (14)
witSetAppData		Use HashMap <Problem, V>	Capabilities Not Implemented	18 (40)
witSetBomEntry{Attribute}	Component	set [overloaded]	Setting Attribute Values	7 (14)
witSetBomEntryAppData		Use HashMap <BomEntry, V>	Capabilities Not Implemented	18 (40)

WIT API Function(s)	WIT-J Class	WIT-J Method	Section	Sect # (Pg #)
witSetBopEntry{Attribute}	Component	set [overloaded]	Setting Attribute Values	7 (14)
witSetBopEntryAppData		Use HashMap <BopEntry, V>	Capabilities Not Implemented	18 (40)
witSetDemand{Attribute}	Component	set [overloaded]	Setting Attribute Values	7 (14)
witSetDemandAppData		Use HashMap <Demand, V>	Capabilities Not Implemented	18 (40)
witSetExtOptSoln{Dbl}			Capabilities Not Implemented	18 (40)
witSetMesgFileAccessMode	MessageMgr	setMesgFileAccessMode	Message Control Methods	13 (32)
witSetMesgFileName	MessageMgr	setMesgFileName	Message Control Methods	13 (32)
witSetMesgPrintNumber witSetMesgStopRunning witSetMesgThrowErrorExc			Capabilities Not Implemented	18 (40)
witSetMesgTimesPrint	MessageMgr	setMesgTimesPrint	Message Control Methods	13 (32)
witSetObjectiveList	Problem	setObjectiveList	Quasi-Attribute Methods	12 (30)
witSetOperation{Attribute}	Component	set [overloaded]	Setting Attribute Values	7 (14)
witSetOperationAppData		Use HashMap <Operation, V>	Capabilities Not Implemented	18 (40)
witSetOptInitMethod	Problem	setOptInitMethod	Quasi-Attribute Methods	12 (30)
witSetPart{Attribute}	Component	set [overloaded]	Setting Attribute Values	7 (14)
witSetPartAppData		Use HashMap <Part, V>	Capabilities Not Implemented	18 (40)
witSetSubsBomEntry{Attribute}	Component	set [overloaded]	Setting Attribute Values	7 (14)
witSetSubsBomEntryAppData		Use HashMap <Substitute, V>	Capabilities Not Implemented	18 (40)
witSetWit34Compatible witShutDownExtOpt			Capabilities Not Implemented	18 (40)
witShutDownHeurAlloc	Problem	shutDownHeurAlloc	Wrapper Methods	10 (22)
witStartExtOpt			Capabilities Not Implemented	18 (40)
witStartHeurAlloc	Problem	startHeurAlloc	Wrapper Methods	10 (22)
witStochImplode	Problem	stochImplode	Wrapper Methods	10 (22)
witWriteCriticalList	Problem	writeCriticalList	Wrapper Methods	10 (22)
witWriteData	Problem	writeData	Wrapper Methods	10 (22)

WIT API Function(s)	WIT-J Class	WIT-J Method	Section	Sect # (Pg #)
witWriteExecSched	Problem	writeExecSched	Wrapper Methods	10 (22)
witWriteReqSched	Problem	writeReqSched	Wrapper Methods	10 (22)
witWriteShipSched	Problem	writeShipSched	Wrapper Methods	10 (22)

23. List of Code Changes

The following is a chronological list of all of the documented changes that have been made to WIT-J since its first release.

1. An approach to thread safety has been implemented by the use of a new class: **`ThreadedObject`**. See Section 16: Thread Safety (page 37).
2. The `Attribute` methods `requiresMaterialPart()` and `requiresStochMode()` have been removed and replaced with a new `Attribute` method, `isValidFor()`.
3. Access to WIT's new "multiple objectives mode" has been added. This includes 5 new `Attributes`:
   - `multiObjMode`
   - `objectiveListSpec`
   - `currentObjective`
   - `objectiveSeqNo`
   - `multiObjTol`

   It also includes 2 new `Problem` methods:
   - `setObjectiveList`
   - `getObjectiveList`
4. The following `Attributes` have been added:
   - `selectionRecovery`
   - `leadTimeUB`
   - `boundedLeadTimes`
   - `modHeurAlloc`
5. Access to WIT's new capabilities relating to CPLEX has been added. This includes 11 new `Attributes`:
   - `coinEmbedded`
   - `coinSelected`
   - `cplexEmbedded`
   - `cplexSelected`
   - `cplexParSpecName`
   - `cplexParSpecIntVal`
   - `cplexParSpecDblVal`
   - `cplexStatusCode`
   - `cplexStatusText`
   - `cplexMipBound`
   - `cplexMipRelGap`

   It also includes 5 new `Problem` methods:
   - `addIntCplexParSpec`
   - `addDblCplexParSpec`
   - `getIntCplexParSpec`
   - `getDblCplexParSpec`
   - `clearCplexParSpecs`
6. Class `GlobalAspect` has been removed. Its methods have been moved to class `Problem`, which is now a `Component` class. All `Attributes` that applied to class `GlobalAspect` now apply to class `Problem`.

7. Method `getCriticalList` was added to class `Problem`.
8. Method `getPgdCritList` was added to class `Problem`.
9. Method `getBelowList` was added to class `Part`.
10. Method `getExpCycle` was added to class `Problem`.
11. Method `getSortedParts` was added to class `Problem`.
12. Method `getSortedOperations` was added to class `Problem`.
13. Method `copyComponentData` was added to class `Component`.
14. The following Attributes were removed due to the removal of the corresponding attributes from WIT for the removal of COIN from WIT:

- `coinEmbedded`
- `coinSelected`
- `cplexSelected`

15. The following methods were added to class `Problem`:

- `getNPartsCreated`
- `getNDemandsCreated`
- `getNOperationsCreated`

16. The following `Attributes` were added:

- `pipEnabled`
- `pipRank`

17. The following methods were added to class `Operation`:

- `getConsVolPip`
- `getCoExecVolPip`
- `getExecVolPip`
- `getProdVolPip`
- `getSideVolPip`
- `getSubVolPip`
- `getSupplyVolPip`

18. The names of the `Attributes` were changed to all upper case, in order to adhere to standard practice for the names of constant fields in Java. The following method was added to class `Attribute`::

- `getWitName`

19. The names of the `asAttribute` method of class `Attribute <V>` was changed to `asAttOfType`.
20. The `Component` methods for setting and retrieving the values of `Attributes` were revised for compatibility with Java 7.0. They are now:

- `set`
- `setVectorToScalar`
- `setBoundSet`
- `setBoundSetToScalars`
- `get`
- `getVector`
- `getBoundSet`

21. The methods for retrieving the default values of Attributes were revised for compatibility with Java 7.0 and moved to class Attribute <V>. They are now:

- getDefaultValue
- getDefaultBoundSet

References

Bloch, J. 2008. *Effective Java, Second Edition.* Addison-Wesley. ISBN: 0-321-35668-3

IBM. 2012. Watson Implosion Technology, User's Guide and Reference. Release 8.0. Available on GSA at:

`/gsa/yktgsa/projects/w/wit/doc/wit-guide/wit.pdf`

Liang, S. 1999. *The Java Native Interface: Programmer's Guide and Specification.* Addison-Wesley. ISBN: 0-201-32577-2

Wittrock, R. 2012. An Introduction to WIT: Watson Implosion Technology (Second Edition). IBM Confidential Research Report RC 25331. Available on GSA at:

`/gsa/yktgsa/projects/w/wit/doc/intro-wit/intro-wit-2.pdf`