RuleWizard User's Guide Table of Contents

Introducing RuleWizard

Creating Custom Coding Standards

RuleWizard GUI

Reference Guide

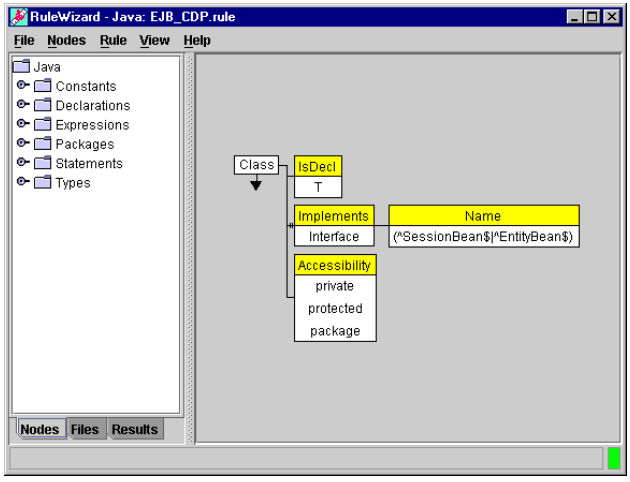
Index

Welcome!

Welcome to RuleWizard, a feature that allows you to create custom coding standards. Jtest can automatically enforce any rule created in RuleWizard.

By allowing you to easily create and enforce standards that are perfectly tailored to your personal, project, team, and company needs, RuleWizard and Jtest provide the most effective and efficient error prevention solution available.

With RuleWizard, you create custom rules by graphically expressing the pattern that you want Jtest to look for when it parses code during static analysis. Rules are created by selecting a main "node," then adding additional elements until the rule expresses the pattern that you want Jtest to check for. Rule elements are added by pointing, clicking, and entering values into dialog boxes. No knowledge of the parser is required to write or modify a rule.



Welcome!

Contacting ParaSoft

ParaSoft is committed to providing you with the best possible product support for RuleWizard. If you have any trouble installing or using RuleWizard, please follow the procedure below in contacting our Quality Consulting department.

- Check the manual.
- Be prepared to recreate your problem.
- Know your RuleWizard version. (You can find it in **Help> About**).
- If the problem is not urgent, report it by e-mail or by fax.
- If you call, please use a phone near your computer. The Quality Consultant may need you to try things while you are on the phone.

Contact Information

- **USA Headquarters**

  Tel: (888) 305-0041

  Fax: (626) 305-9048

  Email: quality@parasoft.com

  Web Site: http://www.parasoft.com

- **ParaSoft France**

  Tel: (33 1) 64 89 26 00

  Fax: (33 1) 64 89 26 10

  Email: quality@parasoft-fr.com

- **ParaSoft Germany**

  Tel: +49 (0) 78 05 95 69 60

  Fax: +49 (0) 78 05 95 69 19

  Email: quality@parasoft-de.com

- **ParaSoft UK**
  Tel: +44 (020) 8263 2827
  Fax: +44 (020) 8263 2701
  Email: quality@parasoft-uk.com

Overview: How to Create a Rule

When you create a rule, your goal is to graphically express the pattern that you *do not* want to appear in your code. When Jtest enforces a rule, it searches for instances where the specified pattern occurs, then flags any violations that it finds.

To open RuleWizard

1. In either Jtest UI, open the Global Test Parameters window by clicking the **Global** button.
2. In the Global Test Parameters window, go to **Static Analysis> Rules> User Defined Rules**.
3. Right-click **User Defined Rules** and choose **Add/Edit Rules (RuleWizard)** from the shortcut menu.

Or,

- In either Jtest UI, right-click the **Rules** button and choose **Launch RuleWizard**.

When the RuleWizard GUI opens, you will see two main panels. The Node tab on the left side of the GUI contains the nodes that you can use as rule subjects.The right side of the GUI is the area where your rule patterns will be displayed.

There are two ways to create a rule: manually and using the Auto-Create feature.

Manually

Creating a rule generally involves the following steps:

1. Using plain English, define the rule that you want to create and enforce.

   For example: "Begin class names with an uppercase letter."

2. Express this concept in terms of RuleWizard elements.

- Create the parent rule node that is the subject of your rule.

    a. In the Node tab, select the node that is the subject of your rule.

    Tip: For a description of a certain node, right-click the node, and choose **View Documentation** from the shortcut menu that opens.

    If you can't find the node you want, right-click inside the white space and choose **Find** from the shortcut menu that opens.

    b. Right-click the selected node, then choose **Create Rule** from the shortcut menu. A rule node will appear in the GUI's right panel.

- Add further qualifications to your node until it fully expresses your rule.

    a. Right-click any of your rule's nodes. All available options for the chosen node will be displayed in the shortcut menu that opens. Any options that are not programming elements or concepts are explained in the section "RuleWizard Commands" on page 69.

    b. Choose a command from the shortcut menu. Depending on the command that you choose, RuleWizard will add a rule element, modify a rule element, or open a dialog box that lets you add or modify a rule element.

    c. If you want to continue adding to and modifying the rule, you can do so by right-clicking any rule node or rule element, then choosing one of the available commands.

- Determine what error message will be displayed when this rule is violated.

a. Create an output arrow by right-clicking a rule node (the placement of the output arrow determines what line number is used for the output message; to have the line number of node C included in the output message, place the output arrow on node C), then choosing **Create Output** from the shortcut menu. The Customize Output window will open.

b. In the Customize Output window, enter a brief explanation of the violation.

c. Click **OK**.
An output arrow will then be added to your rule.

The rule is complete once it:

- Expresses the pattern that you want Jtest to search for, and
- Contains an output arrow and message.

After you customize this rule's properties (you must at least enter a rule header) and save the rule, you can enforce it with Jtest.

Auto-Create

To use Auto-Create, simply right-click inside the white space of the Node tab and choose **Auto-Create Rules** from the short-cut menu. Then in the RuleWizard Automatic Creation window type the code you do not want to appear and click **OK**. RuleWizard automatically generates the nodes and conditions for you. Then you can customize the rule's properties, save it, and enforce it with Jtest.

For more information see “Automatic Rule Creation” on page 43.

Searching for Nodes

To search for a node, simply right-click inside the white space of the Dictionary tab and choose **Find** from the short-cut menu. Then, in the Dictionary window that opens, type the name of the node you want. RuleWizard will search through the nodes and highlight the node that matches what you type. Click **Next** to go to the next node that matches

the text you typed. Click **Clear** to clear the field. Click **Select** to have RuleWizard keep that node highlighted as you continue your search.

Tips

General

- Remember that you are trying to express a pattern that constitutes a violation of the rule.
- As you create your rule, look at the status bar for tips on creating a valid rule. The color of the bar in the right side of the status bar indicates whether or not a rule is valid: a red bar indicates that the rule is not yet valid; a green bar indicates that the rule is valid. The messages in the status bar tell you how to make an invalid rule valid.
- Be sure to include at least one output arrow in each rule. If a rule does not have an output arrow, Jtest will not report an error if this pattern is found in the code under test. To include an output arrow, right-click a rule node (the placement of the output arrow determines what line number is used for the output message; to have the line number of node C included in the output message, place the output arrow on node C), then choose **Create Output** from the shortcut menu.
- The relationship between all rule branches is AND unless you indicate otherwise. If you want to express a different relationship between branches (for example, OR, NAND, or NOR relationships), you can do so by adding or changing logic components. For more detail on logic components, see “RuleWizard Commands” on page 69.
- To undo a command, simply right-click the gray portion of the workspace and choose **Undo**.
- Be sure to enter a header when you customize this rule's properties; rules without headers are not valid.
- Any node (such as **bool Constant**) or folder of nodes (**Constants**) can be used as a rule node.

- To view a description of a node in the Nodes tab, right-click the node that you want more information about, then choose **View Documentation** from the shortcut menu.
- Be aware that RuleWizard is order-specific. If your rule has more than one child node, you can move the child up or down one position by right-clicking the vertical line common to all children, then choosing **Move up one** or **Move down one** from the shortcut menu.

Expressions and Regular Expressions

- Use expressions to match number values, and regular expressions to match strings of text. For guidelines on using expressions and regular expressions, see “Expressions and Regular Expressions” on page 76.

Rule Conditions

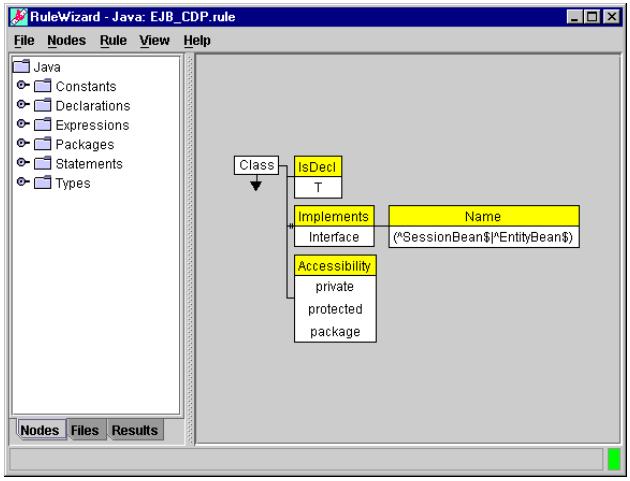
- To create a rule with a pattern that involves a union, intersection, exclusive-or relationship, or difference of multiple nodes, see “Working With Node Sets” on page 44.
- To create a rule condition that restricts the number of a certain element that can appear in a block (like a file or a class), create a collector to track the number of instances of that pattern, then use **Count** to specify the number of instances that constitutes a violation. For information on determining exactly how “counts” are calculated, see “Working With Node Sets” on page 44.
- To create a rule condition about the code element that contains the parent node, use **Context**.
- To create a rule condition about the parent node’s condition statement (**if**, **else**, **init**, **increment**, **for**, **with**, **while**, **switch**, and **do while**), use **Condition**.
- To create a rule condition about the code element that is a subnode of the parent node, use **Body**.
- To indicate whether or not Jtest searches nodes recursively when it searches for the rule condition, use **Indirect Check** and **Direct Check**.

Indirect Check is the default setting; to use a **Direct Check**, right-click the node whose checking type you want to change, then choose **Direct Check** from the shortcut menu.

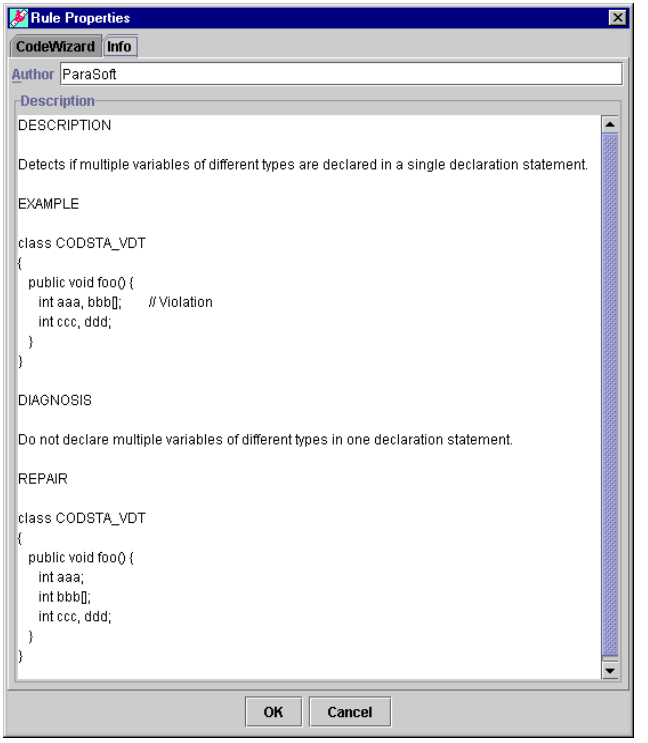
Exploring Example Rules

One of the best ways to familiarize yourself with RuleWizard is to look at how example rules are constructed. You can find a set of example rule files in `<jtest installation directory>/brules`. These rule files are the source for many of Jtest's built-in static analysis rules.

To open a rule file, choose **Rule> Open Rule**, then select one of the available .rule files from the file chooser.



Once the rule is open, you can view the rule description by choosing **Rule> Properties**.



Note: You can also open example rule files by:

1. Clicking **Global** in one of Jtest's UIs.
2. Opening **Static Analysis> Rules> Built-in Rules**.

Right-clicking the rule that you want to open and choose **View RuleWizard Rule** from the shortcut menu.

This menu item is only available if the rule is implemented in RuleWizard.

How to Customize Rule Properties

To specify such rule properties as Rule ID, header, severity, author,and description, choose **Rule> Properties**, then enter the desired properties in the Rule Properties panel.

How to Save and Enable a Rule

In order to enforce a rule, you must save it. To save your rule, choose **Rule> Save** or **Rule> Save As**. This command will invoke a file chooser in which you can specify the rule's filename and path. A .rule extension is automatically assigned to each rule. If you do not use this exact extension, Jtest will not load your rules properly.

After you have saved your rule(s), you can exit RuleWizard.

To enable your rule:

1. Click **Global** in either of Jtest's UIs.
2. In the Global Test Parameters window, go to **Static Analysis> Rules> User Defined Rules**, then right-click **User Defined Rules**. A shortcut menu will open.
3. Choose **Reload Rules** from the shortcut menu that opens.

Your rule will then be enabled; it will be contained in in the **Static Analysis> Rules> User Defined** branch of the Global Test Parameters tree.

How to Automatically Enforce Your Custom Coding Standards

To have Jtest enforce a custom coding standard that you created, saved and enabled:

1. Verify that Jtest is configured to enforce the custom coding standards you want it to enforce. Both the rule and its severity level must be enabled.
    - To do this, open the Global Test Parameters window, then go to **Static Analysis> Rules> Severity Levels** and **Static Analysis Rules> User Defined Rules**. Enable all severity level(s) corresponding to the rule(s) you want to enforce. (Each rule's severity level is appended to its label in the **User Defined Rules** branch). Also, verify that the option that corresponds to the appropriate rule is enabled.
2. Run Jtest as normal.

Enabling and Disabling Rules

If a custom rule is not relevant to a particular situation, you might want to suppress the reporting of this rule's violations. Rules created in RuleWizard can be suppressed in the same way that built-in Jtest rules are suppressed. For information about suppressions, refer to the Jtest User's Guide.

When you suppress a rule, Jtest checks for violations, but does not report them. If you do not want violations of a particular rule reported under most circumstances, you may improve testing performance by disabling the rule, then enabling the rule only when you want to enforce it.

If you never enabled a rule, the rule is already disabled.

To disable or enable a rule:

1. Open the Global Test Parameters window by clicking the **Global** button.
2. Go to **Static Analysis> Rules> Static Analysis Rules> User Defined Rules.**
3. To disable a rule, check the box associated with the rule you want to disable (rules are grouped by Rule ID and listed by Rule Header).

   To enable a rule, check the box associated with the rule you want to enable.

Tutorial: Creating and Enforcing Custom Coding Standards

This tutorial will walk you through the steps required to compose and automatically enforce three rules:

1. Instance fields should begin with an underscore (_) (Lesson 1)
2. Assignment within IF statements (Lesson 2)
3. Documentation for methods with over 17 lines

Getting Started

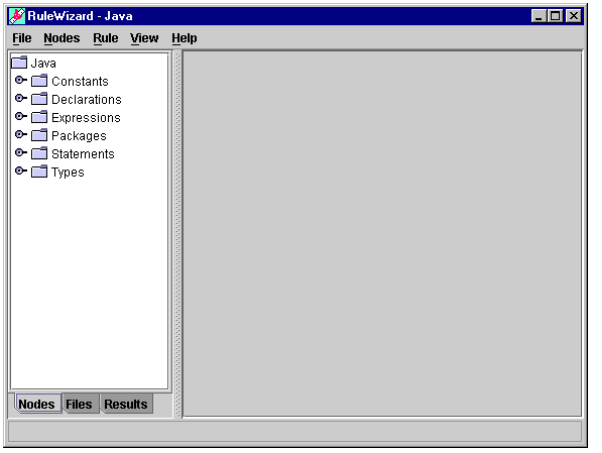
To open RuleWizard:

1. In the Jtest UI, open the Global Test Parameters window by clicking the **Global** button.
2. Go to **Static Analysis> Rules> User Defined Rules**.
3. Right-click **User Defined Rules** and choose **Add/Edit Rules (RuleWizard)** from the shortcut menu.

Or,

- In either Jtest UI, right-click the **Rules** button and choose **Launch RuleWizard**.

When you first open RuleWizard you will see the following GUI:



The nodes with which you build your rules are displayed on the left pane of the GUI. The gray pane on the right of the GUI will be your workspace for composing rules.

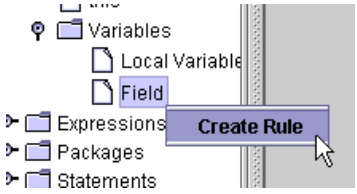
Lesson 1: Begin Instance Fields with Underscore

This rule will report a violation when instance fields do not begin with an underscore.

Designing the Rule Pattern

Creating the Parent Node

Whenever you create a rule, the first thing you need to do to is right-click the node that you want to be your rule's main subject, then select **Create Rule** from the shortcut menu. To start composing this rule, open **Declarations> Variables**, right-click the **Field** node, then choose **Create Rule** from the shortcut menu.

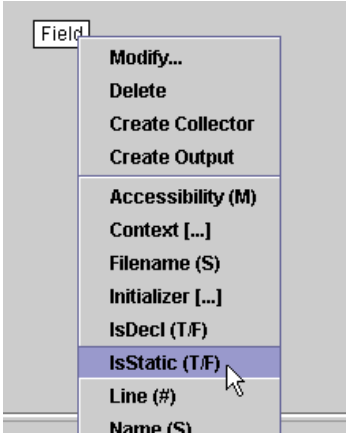


After you choose **Create Rule**, you will see the following parent rule node in the right pane of the GUI:

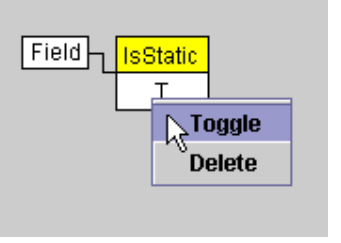
We now have the basic building block for a rule about fields.

Adding Further Qualifications to the Parent Rule Node

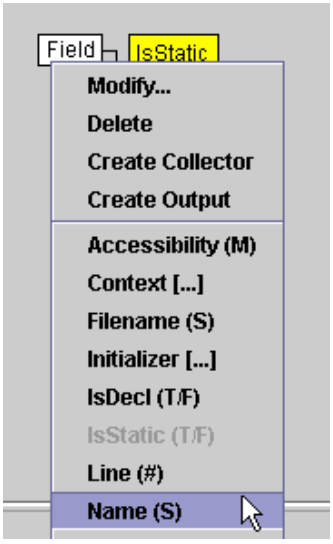
To specify that we want this rule to check instance (non-static) fields, first add the IsStatic property by right-clicking the **Field** rule node and choosing **IsStatic** from the shortcut menu that opens. This says to check if the field is static. This is not what we want to check for, so we will need to modify this node.



To have Jtest check if the field is not static, right-click the **IsStatic** box and choose **Toggle**. This will change this rule condition to say “if an instance variable is present.”

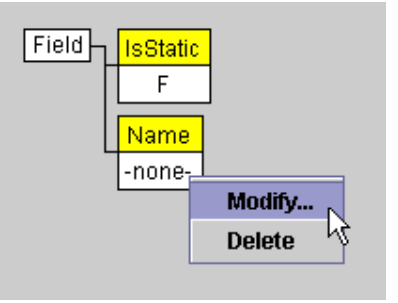


To specify that this rule will be about naming conventions, right-click the **Field** rule node and choose **Name** from the shortcut menu that opens.



A rule node with the content **Name: -none-** will now be attached to your parent rule node.

To continue developing the rule, right-click the **Name** rule node, then choose **Modify** from the shortcut menu.



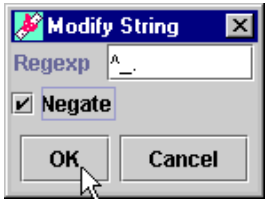
The Modify String window will then open.

In the **Regexp** field of the Modify String window, specify what value Jtest should look for in the name of the instance variable.

- If you want Jtest to report an error if a certain value *is not* present in the code, (as we do in our example), enter the value that you want to require the presence of, then check the **Negate** check box. This tells Jtest to report an error if the specified value is *not* present.
- If you want Jtest to report an error if a certain value *is* present in the code, enter the value that you do not want to appear in your code, then leave the **Negate** check box empty. This tells Jtest to report an error if the specified value *is* present.
- In our example, we want the Name value to begin with an underscore. Thus, we would enter

  ```
  ^_.
  ```

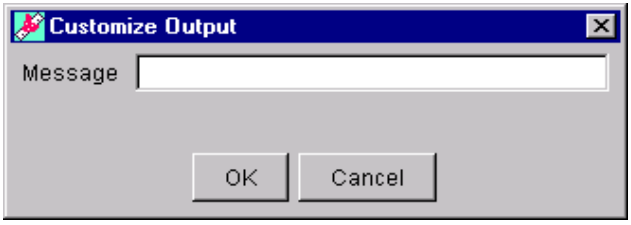
  in the **Regexp** field, and check the **Negate** check box. (The "^" indicates the beginning of an expression and the "." matches any single character).



After you have typed this value and checked the check box, click **OK**.

Specifying an Error Message

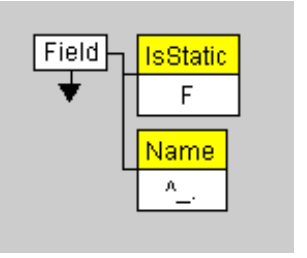
Finally, we need to specify what text Jtest should print when this rule is violated. The first step in doing this is right-clicking the parent rule node (here, the **Field** rule node), then choosing **Create Output** from the shortcut menu.This action will invoke the following Customize Output window:



In the Customize Output window, enter the message that you want Jtest to deliver when this rule is violated. In this example, you might enter “Invalid field name: $name”. When this message is reported by Jtest, $name will be replaced by the actual name of the field.

Click **OK**.

Your rule should now look like this:

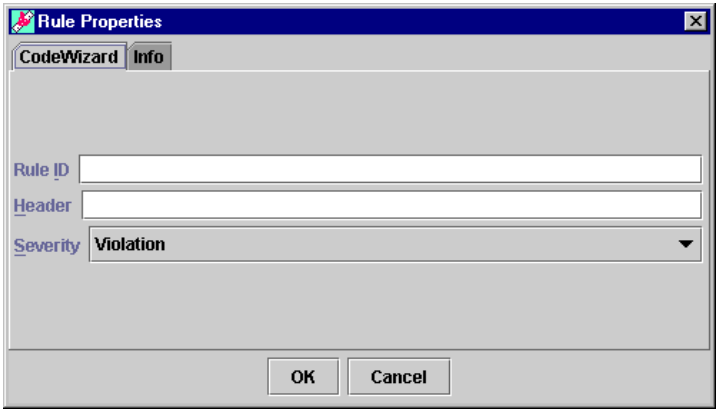


Your rule now tells Jtest to report the specified error message when an instance variable's name does not begin with an underscore. Your rule is now complete. After you customize this rule's properties and save it, Jtest will be able to enforce it.

Customizing Rule Properties

Rule properties can be customized via the Rule Properties panel. To access this panel, choose **Rule> Properties**.

You will then see the Rule Properties panel.



This panel lets you determine the rule's properties. In the **CodeWizard** tab, enter...

- **a rule ID:** the unique ID you want to assign to this rule. If you want Jtest to organize your custom rules into categories, use the following format:

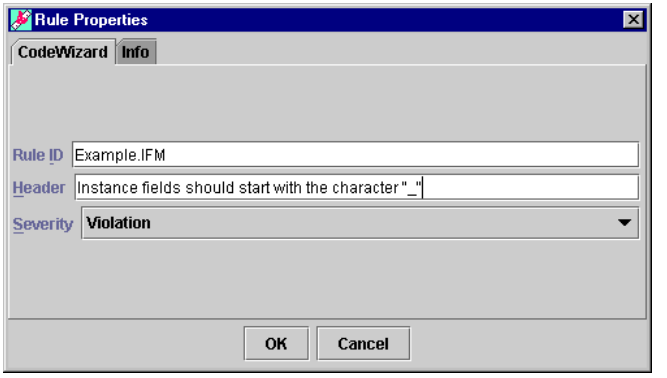
  category.id

  For example, you could use “Example.IFM”

- **a header:** the name you want Jtest to assign to this rule. For this example, you could enter

  Instance fields should start with the character “_”

Next, choose the rule's severity (the severity category in which Jtest will classify the rule). This rule should be categorized as a Violation.



Finally, click the **Info** tab, and enter the name of the rule's author (your name and/or development group) and a description of the rule. (This description will be displayed when users choose **View Rule Description** within Jtest).

When you have entered all of these values, click **OK** to close this panel.

For more information on any of the fields in the Rule Properties panel, see the Rule Properties Panel page.

Saving and Enabling Your Rule

Before you begin composing another rule, or before you exit the program, you will want to save your rule (Jtest only enforces rules that have been saved).

To save your rule, choose **Rule> Save** or **Rule> Save As**. This command will invoke a file chooser in which you can specify the rule's filename and path. Be sure to give each rule you save a .rule extension. If you do not use this exact extension, Jtest will not load your rules properly. Also, be sure to save your rules within the default directory (<Jtest installation directory>/jrules). If a rule is not contained in this directory, Jtest will not enforce it.

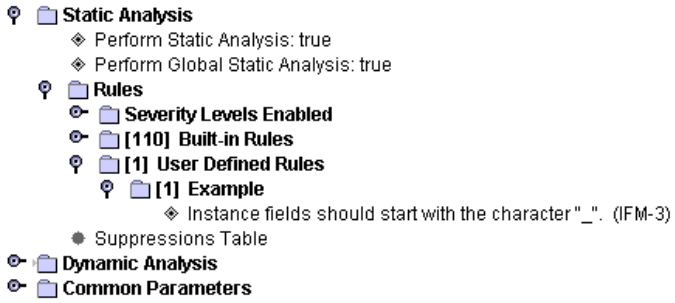
After you save your rule, exit RuleWizard. To enable your rule:

1. Click **Global** in either of Jtest's UIs.
2. In the Global Test Parameters window, go to **Static Analysis> Rules> User Defined Rules**, then right-click **User Defined Rules**. A shortcut menu will open.
3. Choose **Reload Rules** from the shortcut menu that opens.

Your rule will then be enabled.

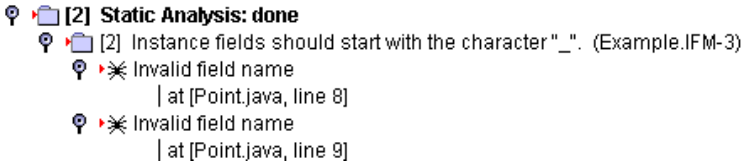
Enforcing Your Rule Automatically

If you look at the Global Test Parameters tree's **Static Analysis> Rules> User Defined Rules** branch, you will see that Jtest automatically added and enabled the rule you just created.



This rule will now be included in the set of rules Jtest applies to your class or classes during static analysis.

To see how this rule is used, test the Point class (located in <Jtest installation directory>/examples/static/userdef). Two errors will be reported for this test.


[2] Static Analysis: done
[2] Instance fields should start with the character "_". (Example.IFM-3)
Invalid field name
at [Point.java, line 8]
Invalid field name
at [Point.java, line 9]

Lesson 2: Assignment Within an IF Statement

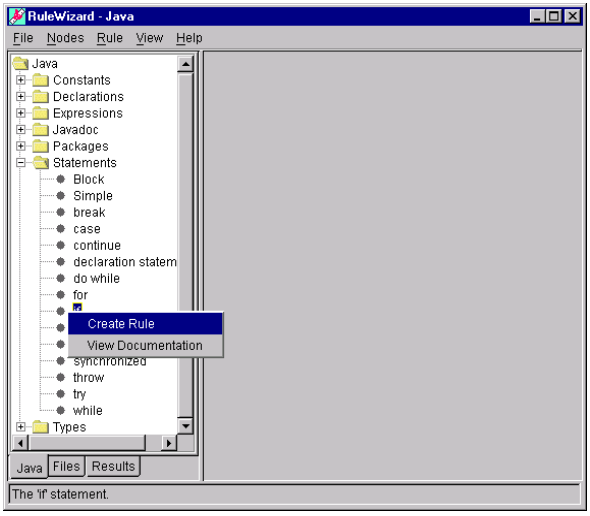
We will now demonstrate how to build a rule that flags instances where assignment is used in IF statement condition. While using assignment within if statement condition is legal (so it won't be caught by a compiler), developers that use a=b in IF statement condition usually intended to use a==b, but made a typographical error. In these cases, using assignment would prevent the code from functioning as intended.

Designing the Rule Pattern

Creating the Parent Node

First, open RuleWizard (if it is not already open).

To start creating this rule, open **Statements**, right-click the **if** node, then choose **Create Rule** from the shortcut menu.

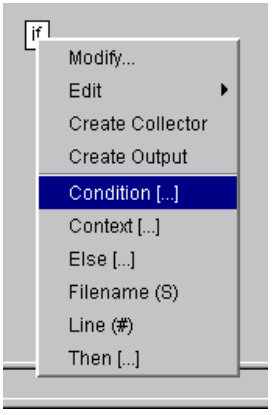


After you choose **Create Rule**, you will see the following parent rule node in the right pane of the GUI:

We now have the basic building block for a rule about IF statements.

Adding Further Qualifications to the Parent Rule Node

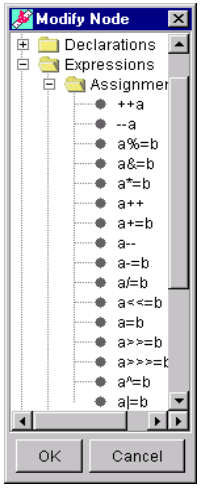
To specify that you want to write a rule about something used in IF statement condition, right-click the **if** rule node and choose **Condition** from the shortcut menu.



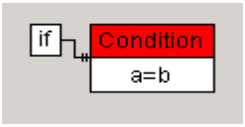
This will create a Condition line and a Condition box that contains the value -none-.

To specify what value you do not want used in if statement condition, right-click the Condition box, then choose **Modify** from the shortcut menu.

The **Modify Node** window will open. Because you want to specify that you do not want assignment used in IF statement condition, choose the assignment node (**a=b**) by opening **Expressions> Assignment**, clicking the **a=b** node, then clicking the **OK** button.



Your rule should now look like this:

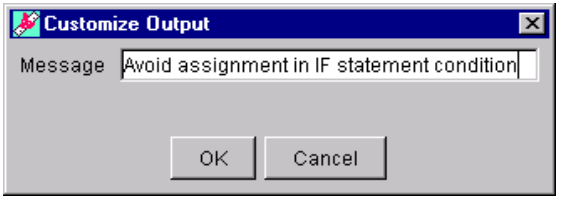


At this point, the rule says to look for instances where assignment is used in IF statement condition. This is what we want the rule to look for, so we will stop modifying the rule and begin specifying output.

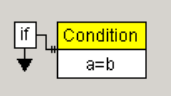
Specifying an Error Message

We now need to specify what text Jtest should print when this rule is violated. The first step in doing this is right-clicking the parent rule node (here, the **if** rule node), then choosing **Create Output** from the shortcut menu.

This action opens the Customize Output window. Enter the message that you want Jtest to deliver when this rule is violated. In this example, you might enter "Avoid assignment in IF statement condition." After you have entered a message, click **OK**.



Your rule should now look like this:



Your rule now tells Jtest to report the specified error message when a developer uses assignment in IF statement condition. Your rule is now complete. At this point, you may want to customize rule properties such as severity, description, author, etc.; these properties can be specified in the Rule Property panel that is accessible by choosing **Rule> Properties**. You should at least specify a header and Rule ID so that Jtest knows how to classify your rule and report rule violations.

Saving and Enabling Your Rule

Before you begin composing another rule, or before you exit the program, you will want to save your rule (Jtest only enforces rules that have been saved).

To save your rule, choose **Rule> Save** or **Rule> Save As**. This command will invoke a file chooser in which you can specify the rule's filename and path. Be sure to give each rule you save a .rule extension. If you do not

use this exact extension, Jtest will not load your rules properly. Also, be sure to save your rules within the default directory (<Jtest installation directory>/jrules). If a rule is not contained in this directory, Jtest will not enforce it.

After you save your rule, exit RuleWizard. To enable your rule:

1. Click **Global** in either of Jtest's UIs.
2. In the Global Test Parameters window, go to **Static Analysis> Rules> User Defined Rules**, then right-click **User Defined Rules**. A shortcut menu will open.
3. Choose **Reload Rules** from the shortcut menu that opens.

Your rule will then be enabled.

Enforcing Your Rule Automatically

If you look at the Global Test Parameters tree's **Static Analysis> Rules> User Defined Rules** branch, you will see that Jtest automatically added and enabled the rule you just created.

This rule will now be included in the set of rules Jtest applies to your class or classes during static analysis.

Note: This rule is included with the product; it's under **Possible Bugs** as "Avoid assignment within an 'if' condition."

Lesson 3: Checking for Documentation

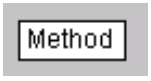
This rule will report a violation when methods with more than 17 statements do not have documentation.

Designing the Rule Pattern

Creating the Parent Node

To start composing this rule, open **Declarations**, right-click the **Method** node, then choose **Create Rule** from the shortcut menu.

You should see the following parent rule node in the right pane of the GUI:



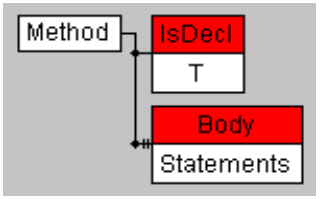
We now have the basic building block for a rule about methods.

Adding Further Qualifications to the Parent Rule Node

To specify that we want this rule to check only declared methods and not references to methods, first add the **IsDecl** property by right-clicking the **Method** rule node (in the workspace panel) and choosing **IsDecl** from the shortcut menu that opens.

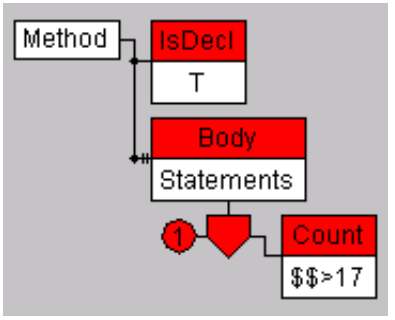
To have Jtest check the statements in the body of the method, right-click the **Method** node and choose **Body**.

Right-clickthe **Body** node and choose **Modify**. Choose the **Statements** node from the list of items in the **Modify Node** window that appears, then click **OK**.

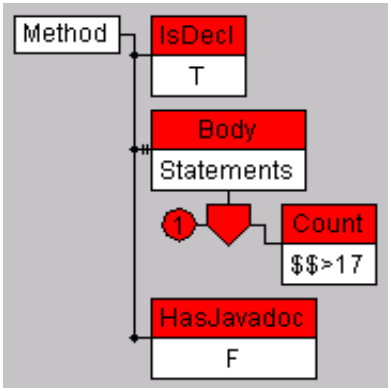


Right-click the **Body** rule node again, then choose **Create Collector** from the shortcut menu. A collector symbol (pentagon) will appear below the **Body** node.

Right-click on the collector and choose **Count #**. To modify the value of **Count**, right-click he **Count** box and choose **Modify**. In the Modify Expression dialog window that opens, enter **$$>17** in the **Expression** field.

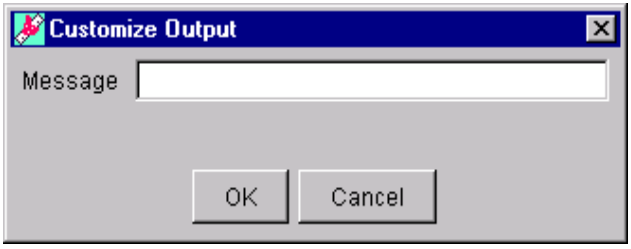


Right-click the **Method** node and choose **HasJavadoc** from the list. Then right-click **HasJavadoc** and choose **Toggle** from the shortcut menu that appears. In our example, we want the method to be flagged if it does not have any Java documentation, so we need to set **HasJavadoc** to false (**F**).



Specifying an Error Message

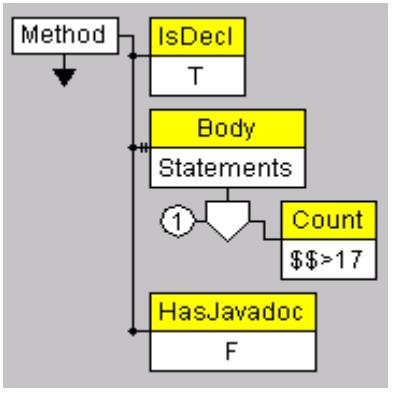
Finally, we need to specify what text Jtest should print when this rule is violated. The first step in doing this is right-clicking the parent rule node (here, the **Method** rule node), then choosing **Create Output** from the shortcut menu. This action will invoke the following Customize Output window.



In the Customize Output window, enter the message that you want Jtest to deliver when this rule is violated. In this example, you might enter, "Missing Javadoc comment for: $name". When this message is reported by Jtest, $name will be replaced by the actual name of the method.

Click **OK**.

Your rule should now look like this:

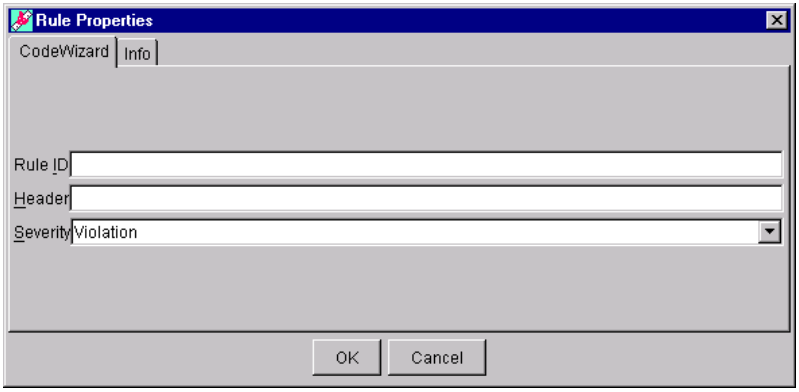


Your rule now tells Jtest to report the specified error message when a method with more than 17 lines does not contain documentation. Your rule is now complete. After you customize this rule's properties and save it, Jtest will be able to enforce it.

Customizing Rule Properties

Rule properties can be customized via the Rule Properties panel. To access this panel, choose **Rule> Properties**.

You will then see the Rule Properties panel.



This panel lets you determine the rule's properties. In the **CodeWizard** tab, enter...

- **a rule ID:** the unique ID you want to assign to this rule. If you want Jtest to organize your custom rules into categories, use the following format:

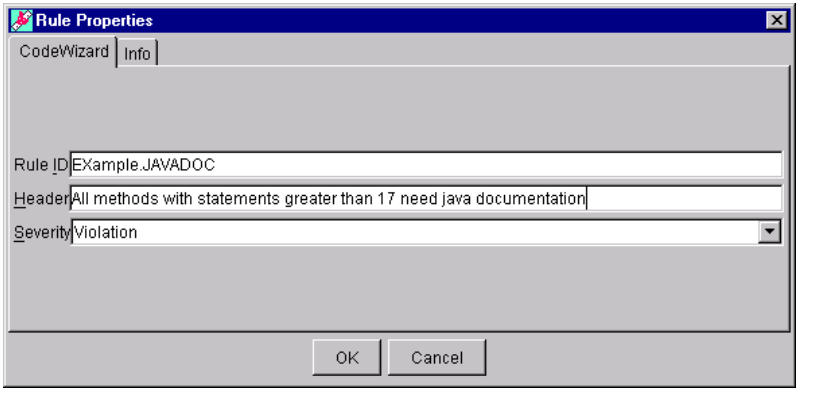
  category.id

  For example, you could use “Example.JAVADOC”

- **a header:** the name you want Jtest to assign to this rule. For this example, you could enter

  All methods with statements greater than 17 need java documentation

Next, choose the rule's severity. This rule should be categorized as a Violation.



Finally, click the **Info** tab, and enter the name of the rule's author (your name and/or development group) and a description of the rule. (This description will be displayed when users choose **View Rule Description** within Jtest).

When you have entered all of these values, click **OK** to close this panel.

For more information on any of the fields in the Rule Properties panel, see the section “Rule Properties Panel” on page 64.

Saving and Enabling Your Rule

Before you begin composing another rule, or before you exit the program, you will want to save your rule (Jtest only enforces rules that have been saved).

To save your rule, choose **Rule> Save** or **Rule> Save As**. This command will invoke a file chooser in which you can specify the rule's filename and path. Be sure to give each rule you save a .rule extension. If you do not use this exact extension, Jtest will not load your rules properly. Also, be sure to save your rules within the default directory (<Jtest installation directory>/jrules). If a rule is not contained in this directory, Jtest will not enforce it.

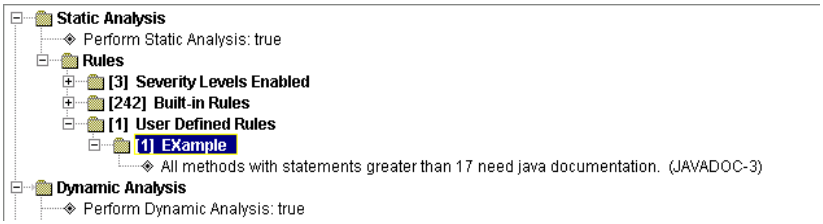
After you save your rule, exit RuleWizard. To enable your rule:

1. Click **Global** in either of Jtest's UIs.
2. In the Global Test Parameters window, go to **Static Analysis> Rules> User Defined Rules**, then right-click **User Defined Rules**. A shortcut menu will open.
3. Choose **Reload Rules** from the shortcut menu that opens.

Your rule will then be enabled.

Enforcing Your Rule Automatically

If you look at the Global Test Parameters tree's **Static Analysis> Rules> User Defined Rules** branch, you will see that Jtest automatically added and enabled the rule you just created.



This rule will now be included in the set of rules Jtest applies to your class or classes during static analysis.

Automatic Rule Creation

One of RuleWizard's most powerful features is the Auto-Create Rules function. You can indicate what code you don't want to appear and RuleWizard does the rest.

To have RuleWizard automatically generate a rule based on dictionary elements:

1. Right-click in the white space of the Node tab and choose **Auto-Create Rules** from the short-cut menu. The Automatic Creation window will open.

2. In the RuleWizard Automatic Creation window, type the code you do not want to appear in your program into the **Sample Java** section.
3. Click **Create**.
   The graphical representation of the rule appears with nodes and conditions in the GUI's right panel.
4. Right-click the Output arrow, replace the text, "Replace this with your own text", then click **OK**.
5. Right-click the gray area in the GUI's right panel and choose **Properties**.
6. In the Rule Properties window, type over the "Automatic Rule" text in the **Header** field, and specify any other properties.
7. Click **OK**.
8. Choose **Rule> Save** to save the rule.
9. In the Save As window, type a name for the rule in the **File name** field and click **Save**.

   The rule is saved.

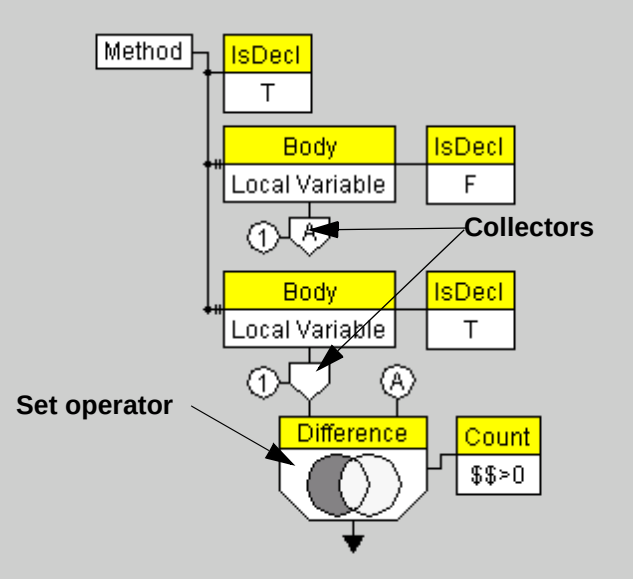
- Once the rule is saved, you can have Jtest enforce it. For more information, see "How to Automatically Enforce Your Custom Coding Standards" on page 15

Working With Node Sets

About Node Sets

When creating complex rules, you may want to create conditions that depend on specific attributes or relationships between multiple nodes. For example, if you have multiple nodes in a rule condition, you may want to specify exactly where Jtest starts and stops counting the number of “hits” (instances where the specified conditions are met). Or, you may want a rule to check if the total number of “hits” in two different sets of nodes is greater than 0. In such cases, you would create a rule that includes one or more set components.

Set components are a category of components that represent sets of nodes. These components include collectors and set operators. Collectors let you restrict a node or node set’s quantity. Set operators let you specify a relationship between two or more set components (for example, they could be used to create a rule that checks that the total number of hits in two rule conditions is less than 1).



Using Set Operators to Specify Relationships Between Set Components

You can use set operators to create rule segments that collect the number of hits of a specific relationship between two set components. For example, you could use set operators to create a rule segment that collects the number of hits of the pattern of nodes that are in either set component A, or in set component B.

Set operators are rule elements that indicate two things:

- Which two set components you want to represent a relationship between.

- What type of relationship you want to establish between the two associated set components.

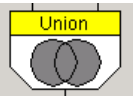
Because set operators establish relationships between set components, they must be connected to one set component (the set component that the set operator is attached to) and reference another set component (the “operand” of the set operator; i.e., the other set component that this set operator works with). Set operators can be used to establish four types of relationships:

- a union of two set components
- an intersection of two set components
- an exclusive-or relationship between two set components
- a difference between two set components

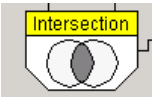
A union is the set of nodes that are:

- in the attached collector, or
- in the operand, or
- in both the attached collector and the operand.

For example, if A represents a set component that contains nodes xx, yy, and zz, and B represents a set component that contains nodes ww, xx, and yy, a union between A and B would match xx, yy, zz, and ww.



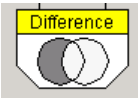
An intersection is the set of nodes that are in both the attached set component and the operand. For example, if A represents a set component that contains nodes xx, yy, and zz, and B represents a set component that contains nodes ww, xx, and yy, an intersection between A and B would match xx and yy.



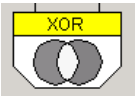
A difference is the set of nodes that are either:

- In the attached set component (the “left” side) but not in the operand (the “right” side), or
- In the operand (the “right” side), but not in the attached set component (the “left” side).

Thus, a right - left difference represents the nodes in the operand, but not the set operator, while a left - right difference represents the nodes that are in the set operator, but not in the operand. For example, if A represents a set component that contains nodes xx, yy, and zz, and B represents a set component that contains nodes ww, xx, and yy, an A-B difference between A and B would match zz, while a B-A difference would match ww.



An XOR (exclusive-or) relationship is the set of nodes that are in either the attached set component or the operand, but not the nodes that are in both. For example, if A represents a set component that contains nodes xx, yy, and zz, and B represents a set component that contains nodes ww, xx, and yy, a XOR between A and B would match zz and ww.

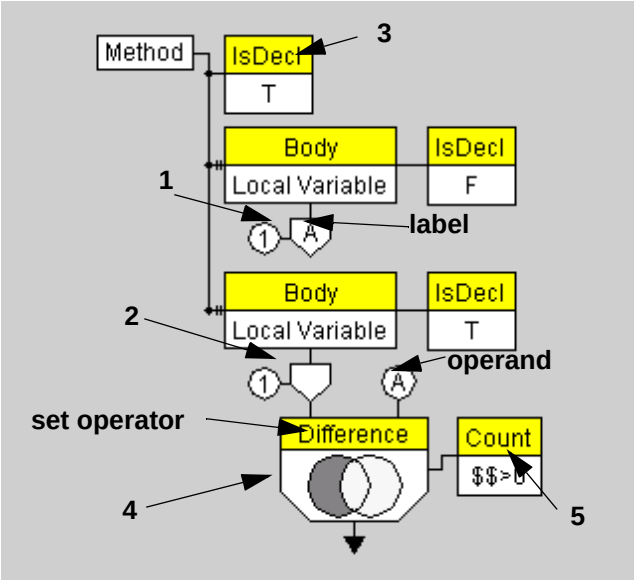


To establish such relationships between two set components (X and Y), you would:

1. Attach a set operator to one set component (X) by right-clicking it and choosing **Set Operator> <desired relation>** from the short-cut menu that opens.
2. Attach a label to the other set component (Y).
3. Specify that you want Y to be the set operator's operand by selecting Y's label as the set component's operand value.

One common reason that you might use set operators is to create a rule that restricts the number of "hits" for a union, intersection, difference, or XOR node set.

The following image is the implementation of "Unused local variable". Please note the following features of this rule:



1. The first collector, labelled A, will collect local variables that are used in non-declaration statements.
2. The second collector will collect local variables that are declared in the method.
3. “IsDecl” represents “Is this declaration?” If it's set to true, it returns true for declaration statements and if it's set to false, it returns true for non-declaration statement.
4. The Difference set operator is used to find the difference of the two node sets.
5. The Count node with $$>0 checks if the Difference set has at least one item. If it has at least one item, Jtest will report an error.

To create a rule that counts and restricts the total number of hits that occur for a specific type of relationship between two node sets:

1. Create a rule condition that contains a set component (such as a collector).
2. Label the set component by right-clicking it, then choosing **Add Label> <label name>** from the shortcut menu that opens. Once you have done this, you can make this component an operand of a set operator.
3. Create another rule condition that contains a set component.
4. Right-click the newly-added set component, then choose **Create Set Operator> <desired type of relationship>** from the shortcut menu that opens. A set operator will then be attached to this set component.
5. Indicate the operand of this set operator by right-clicking the circle on the top right of the set operator and choosing the label of the first set component from the shortcut menu.
6. Restrict the number of "hits" allowed for the specified relationship by right-clicking the set operator, choosing **Count** from the shortcut menu, then (if necessary) modifying the value included in the count node.

Once you have added an output arrow to your rule and specified rule properties, your rule will be complete.

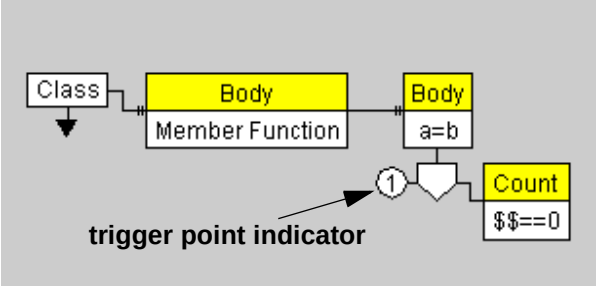
Customizing Counts With Collectors, Counts, and Trigger Points

About Trigger Points

RuleWizard's **Collector** and **Count** commands let you create a rule condition that restricts a node or node set's quantity. To create such a rule condition, you would first create a collector, then right-click the collector and choose **Count**. The collector keeps track of the number of times that the specified pattern is found; **Count** places a condition on what number of instances constitutes a rule violation. When a rule that restricts a node's quantity is enforced, the number of instances of the pattern are collected in the collector, the count is checked, and a violation is reported if the count falls within the parameters specified in the rule.

By specifying when Jtest should empty the collector, you can determine precisely how the count is determined. This is done through the use of "trigger points." Trigger points determine the node at which Jtest starts and stops counting the number of instances that occur. The correct trigger point number to use is determined by counting back from the node to which the collector is attached, to the node at which you want the collector emptied and a violation reported (if the specified pattern is found).

For example, consider the following example:



In this example, a trigger point of 1 indicates to empty the collector after *each member function* is searched. (The **Member Function** node is one node "back" from the **a=b** node to which the collector and trigger point are attached). When enforced, this rule would, for each class, look for a member function in the class' body, then look in that member function for expressions in the form "a=b". The number of expressions that met this criteria would be placed in the collector, the count would be checked, and a violation would be reported if the count were zero. When another member function was parsed, the collector would then be emptied, and this process would be repeated for that member function.

If you changed the trigger point to 2, Jtest would look at all member functions and count the number of matching expressions in *all* member functions before emptying the collector. (The collector accumulates all values found from the **Class** node (2 nodes back from the node to which the trig-

ger point and collector are attached), and the collector is not emptied until it a new class is parsed).

If you had a trigger point of 3 (the highest possible value in this example), Jtest would empty the collector only after the *entire file* was searched. (The collector accumulates all values found from the file node (2 actual and one applied node (the file is an applied node) back from the node to which the trigger point and collector are attached), and the collector is not emptied until a new file is parsed).

Creating a Trigger Point

When you create a collector, it is assigned a number 1 trigger point by default. To change the trigger point value, right-click the trigger point number and choose the desired value from the shortcut menu that opens.

Guidelines for using trigger points:

- To get the maximum value for a particular trigger point, count the number of nodes from the node with the attached collector to the top node of the rule, then add 1.
- You cannot place a trigger point on a node for which a direct check is performed.
- You cannot add an output arrow between the collector and the node/applied node that the trigger point number points to. For example, in the above rule you could place an output arrow at the **Class** node when you have a trigger point of 1 or 2, but for trigger point 3 you would need to attach the output arrow to the collector (rather than to the **Class** node).

Determining the Output Type of a Set Component

If you place an output arrow on a set component, you will be asked to determine when the output “fires” (reports that the pattern has been violated). The available output options for set components are:

- **Hits Output:** Fires on each “hit”, or each node contained in the set. This type of output arrow is placed below the center of the set

operator that it is attached to. When you choose this type of output, the violation output can contain fields of nodes contained in the set operator. For example, if you have a collector that contains variables in Java or C++ and you select a Hits Output, you can use an output message such as “Initialize all variables in constructor. Variable $name is not initialized.” (When this rule is actually violated, the $name variable will be replaced with the name of the uninitialized variable).

- **Trigger Output:** Fires once for the set when the set is triggered at the trigger point. This type of output arrow is placed below the left side of the set operator that it is attached to, or below the trigger point number. When you choose this type of output, the violation output can contain properties of the collector, but not properties of nodes contained in the set operator. Currently, the only variable you can use here is $count.

If your rule includes a collector, you can have your output include the number of “hits” that the collector has accumulated. To do this, enter COUNT(A) (where A is the label of the collector whose count you want reported) in the appropriate output message.

You can also have your output message list the items that a collector has accumulated. To do this, enter LIST(A) (where A is the label of the collector whose “list” you want reported) in the appropriate output message.

These two variables (COUNT and LIST) are called set references.

File Menu

The File menu contains the following commands:

- **Customize Preferences:** Opens the RuleWizard Preferences panel, which allows you to customize such RuleWizard options as rule view and rule file directory.
- **Save Preferences:** Saves the current RuleWizard Preferences.
- **Print:** Prints the contents of the current window. How well this works depends on your Java™ implementation.
- **Exit:** Closes RuleWizard. Any rules that were open will be saved as you exit.

Nodes Menu

The Nodes menu contains the following command:

- **Properties:** Indicates which Node Dictionary you are currently using.

Rule Menu

The Rule menu contains the following commands:

- **New Rule:** Closes the current rule and adds the first node of a new rule.
- **Open Rule:** Opens a saved rule.
- **Close:** Closes the current rule.
- **Save:** Saves the current rule.
- **Save As:** Saves the current rule; lets specify rule name and location.
- **Recent Files:** Provides a shortcut menu of recently opened rules. To re-open a rule, choose it from the menu.
- **Properties:** Allows you to describe the properties of the current rule, including Rule ID, Header, Severity, Author and Description fields.
- **RuleDocs:** Allows you to **Update** and **View** documentation automatically generated by RuleWizard for custom rules.

View Menu

The View menu contains the following commands:

- **Show/Hide status bar:** Displays/hides the status bar at the bottom of the GUI.
- **Show/Hide file viewer:** Displays/hides the file tab.

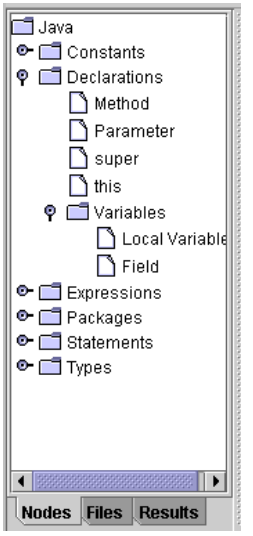
Help Menu

The Help menu contains the following commands:

- **View:** Displays the RuleWizard User's Guide.
- **About...:** Displays RuleWizard's version number.

Nodes Tab

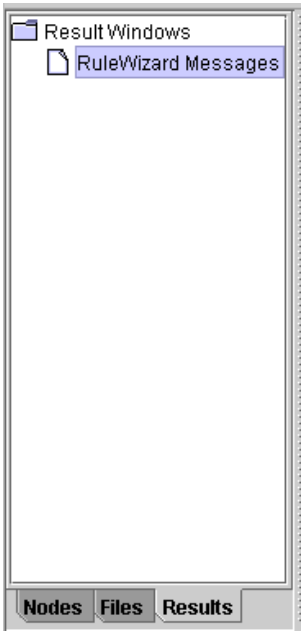
The Nodes tab contains the main elements of Java code. You can use the Nodes tab to start creating rules (by right-clicking the node that you want to be your parent rule node and choosing **Create Rule** from the shortcut menu that opens).



Results Tab

Clicking the Results tab opens the results tree, which displays all messages that RuleWizard has generated. To view the results tree, click the Results tab at the bottom of the left GUI pane. To view RuleWizard Messages, simply click on that category; all results will be displayed in the right GUI pane.

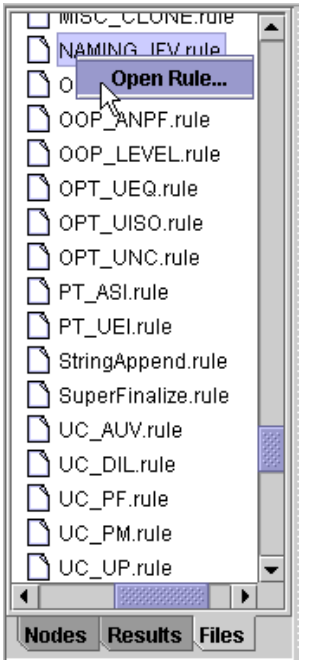
Clicking the RuleWizard Messages results category will allow you to access all messages that have appeared on the status bar. This feature provides you with a convenient way to go back and review a message that is no longer visible. To clear the RuleWizard Messages, right-click that category and then click **Clear** in the shortcut menu.



Files Tab

The Files tab displays the structure of your directories. You can use it to open rule files; to do this, right-click the rule file that you want to open, then choose **Open Rule** from the shortcut menu.

If you do not see the Files tab, you may enable it by choosing **View> Show File Viewer**.



Status Bar

The status bar displays RuleWizard messages, including tips on how to make the rule-in-progress valid. The color of the bar in the right side of the status bar indicates whether or not a rule is valid: a red bar indicates that the rule is not yet valid; a green bar indicates that the rule is valid. The messages in the status bar tell you how to make an invalid rule valid.

Rule Properties Panel

The Rule Properties panel allows you to specify the following rule properties:

- Rule ID
- Header
- Severity
- Author
- Description

To open the Rule Properties panel, choose **Rule> Properties**.

The Rule Properties panel has two tabs:

- **CodeWizard:** Determines properties that will be used within Jtest.
- **Info:** Lets you store general information about the rule.

CodeWizard Tab

The CodeWizard tab allows you to specify the following properties:

- **Rule ID:** Type the category and rule name in the format

  category.name

  Jtest will organize your rules according to category, then according to name. For example, you might want to use a Rule ID such as PROJECT1.ABC
- **Header:** the name assigned to this rule.
- **Severity:** Choose the severity category in which you want Jtest to classify the rule.

  Jtest classifies rules into groups based on the severity of violating the rule:

  - Severe Violation (Level 1)

- Possible Severe Violation (Level 2)
- Violation (Level 3)
- Possible Violation (Level 4)
- Informational (Level 5)

The Severe Violation category should contain rules whose violation will *definitely* result in a bug. Each category below Severe Violation should contain rules whose violations have a progressively lower probability of resulting in an error.

Jtest users can suppress rules according to their severity, so it is important to classify each error appropriately (If a violation of this rule will very likely lead to an error, give it a high severity; if you classify a severe rule as Informational, a violation of that rule may be overlooked by a user suppressing Informational violations.)

Note: If you do not specify the rule's severity, it will be categorized as a Violation (the default setting).

Info Tab

The Info tab allows you to specify the following properties:

- **Author:** Indicates the name of the rule's author.
- **Description:** Contains a description of this rule. This description will be displayed when you choose **View Rule Description** in Jtest

RuleWizard Preferences Panel

The RuleWizard Preferences panel lets you specify RuleWizard options related to:

- Where rules are stored.
- The web browser to be used.

To open the RuleWizard Preferences panel, select **File> Customize Preferences**.

The Rule Properties panel has three tabs:

- **View:** Determines how rules are displayed.
- **Rule Files:** Determines where rule files are stored.
- **Browser:** Sets browser-related options (such as browser used, browser commands, etc.).

View Tab

The View Tab allows you to specify the following preferences:

- **Pixel spacing between components**: Determines the space between rule nodes. Available options include:
    - **Horizontal:** Determines the horizontal spacing between rule nodes.
    - **Vertical:** Determines the vertical spacing between rule nodes.
    - **Reset to defaults:** Returns spacing settings to their default values.

Rule Files Tab

The Rule Files tab lets you configure the following preferences:

- **Rule Directory:** Determines where your rule files are stored. You can either choose the default directory, or specify an alternate directory in the **Other** field.

Browser Tab

The Browser tab lets you configure the following preferences:

- **Browser:** Determines what browser RuleWizard uses. If you want to use a browser that is not listed, select **Other** and enter the path to the browser executable, as well as any arguments that you want to send to the executable.
- **Command:** Determines browser commands such as executable name and any arguments that you want RuleWizard to pass to that browser. If you select a browser name that is provided, RuleWizard will automatically fill in both **Executable** and **Arguments**. If you select **Other**, you can click **Browse** to navigate to the appropriate **Executable** settings. For **Arguments**, enter "%1".
- **Use DDE:** Determines whether or not Dynamic Data Exchange (DDE) lets programs share information. If you select Use DDE, the changes you make to files as you are using RuleWizard will automatically be applied to other programs that use those same files. **Use DDE** is selected by default and may not be disabled for the **Automatic** option in the **Browser** field. It is selected by default but may be disabled for **Netscape Navigator** and **Internet Explorer**. When **Other** is selected in the **Browser** field, **Use DDE** is disabled by default and may not be enabled.

RuleWizard Commands

When you right-click the workspace area around the rule, a shortcut menu containing all available commands will open.

- **Undo:** Undoes the previous command.
- **Paste New Head Node:** Pastes the node from the clipboard as the new head node.

When you right-click a rule node or element, a shortcut menu containing all available commands for that node or element will open. Most commands are programming elements or concepts. The following commands are unique to RuleWizard:

- **Edit:** Lets you use the editing commands; you can cut, copy, delete, or paste the selected node or element.
    - **Cut:** Cuts the selected node from the workspace area and places it on the clipboard.
    - **Copy:** Copies the selected node or element onto the clipboard
    - **Delete:** Deletes the selected node or element from the workspace.
    - **Paste:** Pastes the contents of the clipboard onto the workspace.
    - **Paste As:** This command allows you to paste the contents of the clipboard as a different form. For a copied or cut node this can be as a condition (**Body**, **Context**, etc.). For a copied or cut element, they can be pasted as another element (**IsDecl**, **IsFinal**, etc.).
- **Create Assertion:** Adds an assertion. An assertion specifies the numerical relationship between node collectors. Assertions use the set references MIN, MAX, AVERAGE, and MEDIAN. They are used on set components whose elements are number nodes to compare the values of the counts.
  For information on assertions, see “Working With Node Sets” on page 44.

- **Add Logic Component:** (Only available if logic components are enabled. To enable logic components, change the showLogicComponent variable in <jtest install dir>/bin/lib/Java.cwd from "false" to "true" If logic components are not enabled, AND will always be used).
  Adds a logic component. Logic components let you place AND, OR, NAND, and NOR conditions on rule branches. Available options are:
  - **AND:** Every branch must be true for the expression to return true.
  - **OR:** At least one branch must be true for the expression to return true.
  - **NAND:** At least one branch must be false for the expression to return true. (This is the negation of AND).
  - **NOR:** Every branch must be false for the expression to return true.
- **Change Type:** (Only available if logic components are enabled. To enable logic components, change the showLogicComponent variable in <jtest install dir>/bin/lib/Java.cwd from "false" to "true" If logic components are not enabled, AND will always be used). Changes the type of logic component.
- **Create Collector:** Adds a collector, displayed as a pentagon, to the selected rule node. The collector allows you to place numerical stipulations on the outcome of a rule. Works with the **Count** command (available by right-clicking the collector). The collector keeps track of the number of times a pattern is found; **Count** places a condition on what number of instances constitutes a rule violation.
- **Create Output:** Adds an output arrow to the selected rule node. The output arrow is the essential closing to any rule. An output arrow tells RuleWizard to report an error if the specified pattern is found; if no output arrow is included, no violations of this rule can be reported. The placement of the arrow determines what line number is reported in the error message. For example, if you have a rule with nodes A, B, and C and you attach the output arrow to node C, the line number will reference the line where C

occurs; if you attach the output to node A, the line number will reference the line where A occurs.

If you are adding an output arrow to a set component (a collector or set operator), you must choose between the following two types of output arrows:

- **Hits Output:** Choose this output to indicate that:
    - One output message should be reported each time that the attached node is matched
    - The output message should reference the line number of the node to which the collector is attached. For example, if you have a collector that contains variables in Java or C++ and you select a Hits Output, you can use an output message such as “Initialize all variables in constructor. Variable $name is not initialized.” (When this rule is actually violated, the $name variable will be replaced with the name of the uninitialized variable).

      This type of output can contain variables (such as “$tag”, or “$value”) for nodes contained in the set collector or operator. For example, if you have a collector that contains variables in Java or C++ and you select a Hits Output, you can use an output message such as “Initialize all variables in constructor. Variable $name is not initialized.” (When this rule is actually violated, the $name variable will be replaced with the name of the uninitialized variable).
- **Trigger Output:** Choose this output to indicate that:
    - One output message should be reported each time that the trigger point is matched (for example, if the trigger point references a file, one message will be reported for each file).
    - The output message’s line number should reference the line number of the trigger point node.

This type of output arrow is placed below the center of the set operator that it is attached to.

When you choose this type of output, the violation output can contain properties of the collector, but not properties of nodes contained in the set operator. Currently, the only variable you can use here is “$count”.

- **Indirect/Direct Check:** When an indirect check is performed, Jtest searches for the specified condition in all nodes that have the specified relationship to the given node.
  When a direct check is performed, Jtest searches for the specified condition only in the first node that has the specified relationship to the given node.

- **Create Set Operator:** Creates a set operator. Set operators are used to specify relationships between set components.

  Available options include:

  - **Union:** The set of nodes that are in either the attached set component or the operand, including the nodes that are in both.
  - **Intersection:** The set of nodes that are in both the attached set component and the operand.
  - **Difference:** The set of nodes that are in the attached set component but not in the operand, or the set of nodes that are in the operand but not in the attached set component.
    - **Left minus Right:** The set of nodes that are in the attached set component (to the left of the set operator), but which are not in the operand (as indicated in the circle attached to the right of the set operator).
    - **Right minus Left:** The set of nodes that are in the operand (as indicated in the circle attached to the right of the set operator), but which are not in

the attached set component (to the left of the set operator).

- **XOR:** The set of nodes that are either in the attached set component, the operand, but not in both.

For information on set components and set operators, see “Working With Node Sets” on page 44.

- **Label:** Labels a set component so that it can be used as the operand of a set operator.

RuleWizard commands will be displayed at the top of the shortcut menu; options that pertain to programming elements and concepts are displayed below the menu's line.

Each non-RuleWizard command in the shortcut menu is followed by a symbol that describes the function of the item:

[...] indicates an item which can have either a direct or indirect check.

(S) indicates an item that takes a string input.

[T/F] indicates an item that lets you toggle between true and false inputs.

[M] indicates an item that lets you choose between predetermined input options. For example, Permission [M].

(#) indicates an item that takes a numerical input.

***** indicates a property/node that is available from more than one command. For example, if you can choose the **Body** property from more than one of the available commands, the **Body** command will contain an asterisk.

The commands available depend on which node or rule element was selected. Some of these commands that you may not be familiar with include:

- **Count:** Lets you create a rule condition that restricts a node’s quantity. Must be used in conjunction with the **Create Collector** command (First, create a collector, then right-click the collector and choose **Count**). The collector keeps track of the number of times a pattern is found; **Count** places a condition on what number of instances constitutes a rule violation. For information

on determining exactly how counts are calculated, see “Working With Node Sets” on page 44.

- **Body:** Lets you create a rule condition about the code element that is a subnode of the parent node. (The body of Node A returns a “body” that is A’s subnode; the exact definition of the “body” depends on the node itself).
- **Context:** Lets you create a rule condition about the code element that contains the parent node. (The context of Node A returns the node that contains Node A). For example, if you wanted to create a rule that said “always put X inside of Y,” you would create a parent node for X, use Context to attach Y, create a collector, then use Count to specify that a count of $$==0 constitutes a violation.

  *Note: Some rules can use only **Body**, some can use only **Context**, some can use neither, and some can use both. If you have a choice, choose **Body** because **Body** will result in better performance than **Context**. In many-- but not all-- cases, **Body** and **Context** are inverse operations. For example, an expression can be in the context of a statement, but be contained in the condition (rather than the body) of the statement.*

- **Condition:** Lets you create a rule condition about the parent node’s statement (**if**, **increment**, **for**, **while**, **switch**, and **do while**).

Expressions and Regular Expressions

Expressions

Expressions are used to match values; $$ is used with expressions to indicate a variable. You can enter expressions in the Modify Expression window. A few examples of valid expressions that you could enter in this window include:

Expression	Matches	Example
$$==n	a value equal to n	$$==1 matches values equal to 1
$$<n	a value less than n	$$<100 matches values less than 100
$$>n	a value greater than n	$$>100 matches values greater than 100
$$<=n	a value less than or equal to n	$$<=550 matches values less than or equal to 550
$$>=n	a value greater than or equal to n	$$=>1 matches values greater than or equal to 1

Regular Expressions

Regular expressions are used to match strings. Regular expressions are supported by many languages, including Perl, Python, and Ruby. RuleWizard's regular expressions are similar to those supported by Perl. You can enter regular expressions in **Regexp** fields of Modify String windows. Here are some guidelines for entering regular expressions:

character/ metacharacter	**Matches**	**Examples**
anystring	an occurrence of the string "anystring"	"soft" matches parasoft, software, soften, etc.
.	exactly one non-null character	".at" matches hat, cat, bat, fat, etc., but not at w...ing matches webking, working, but not what a king or wing
?	0 or 1 occurrences of the preceding character	"j?test" matches either jtest or test
*	0 or more occurrences of preceding character	"a*soft" matches asoft, or aaaaasoft; ".*ing" matches webking, waning, wing, what was that thing
+	1 or more occurrences of the preceding character	"a+soft" matches aaaaasoft, or aaasoft, but not asoft

character/ metacharacter	Matches	Examples
[]	matches one occurrence of any character inside the brackets; ^ inverts the brackets metacharacter	“[cpy]up” matches cup, pup, or yup “rule0[1-4]” matches rule01, rule02, rule 03, rule 04 “[^ch]at” matches all 3 letter words ending with “at” except for cat and hat.
[A-Z]	any uppercase letters from A to Z	“[A-Z]” matches any uppercase letter from A to Z
[a-z]	any lowercase letters from a -z	“[a-z]” matches any lowercase letter from a-z
[0-9]	any integer from 0 to 9	“rule[0-9]” matches any expression that begins with “rule” and ends with an integer
{}	like *, but the string it matches must be of the length specified in the braces	“a{2}” matches aa, aaa, aaa, etc. “a{3,}” matches at least 3 occurrences of the preceding character (aaaaaa, or aaaaaaaa, but not aa “a {2,5}” matches between 2 and 5 occurrences of the preceding character (aaa, aaaaaa, but not aa or aaaaaaaaaaaaaaaaaa) “^(a{2})$” matches only aa

character/ metacharacter	Matches	Examples
|	matches the string before the “|”, the string after the “|”, or both	“rulewizard|codewizard” matches rulewizard, codewizard, or both

Additional Tips

- ^ indicates the beginning of a string in parentheses; $ indicates the end of a string in parentheses. Thus, to get an exact match for a string, use the format ^(STRING)$. For example ^(soft)$ would only flag “soft”.
- If you want Jtest to report an error if the expression is detected, leave the Regexp window’s **Negate** check box empty.
  Note: You reach the Regexp window when you choose to modify a string (such as a name value).
- If you want Jtest to report an error if the expression is not detected, check the Regexp window’s **Negate** check box.
- Regular expressions searches are case sensitive by default.
- When using regular expressions, “\" is an escape character that you can use to match a "." , "*", or another character that has a non-literal meaning.

Available Rule Nodes

The nodes that you can use to create your rule are listed below, in the order in which they appear in a fully expanded node tree.

Constants

Nodes which represent values which cannot be changed.

boolean Constant

A node which represents a boolean constant.

Example:

```
public final boolean isHuman = true;  // boolean constant
```

char Constant

A node which represents a character constant.

Example:

```
private final char FIRST_LETTER = 'A';  // char constant
```

int Constant

A node which represents an integer constant.

Example:

```
private final int MIN_AGE = 21;  // int constant
```

long Constant

A node which represents a long integer constant.

Example:

```
long MAX_TEMP = 451L;  // long constant
```

float Constant

A node which represents a floating point constant.

Example:

```
private final float yourNumber = 2.5f; // float constant
```

double Constant

A node which represents a double floating point constant.

Example:

```
public final double myNumber = 314159e-5d;  // double constant
```

null

A node which represents the built-in value null.

Example:

```
private StringTokenizer tokenizer = null;  // null
```

String Constant

A node which represents a String constant.

Example:

```
public final String name = "Mercury";  // String constant
```

Declarations

Represents a user-defined data type or method.

The engine uses the IsDecl property to distinguish between the actual declaration and references to the declaration.

Method

A method (function) declaration.

Example:

```
public int myMethod() { // Method
    int myValue = 17;
    return myValue;
}
```

Parameter

A declaration which is in the parameter list of a method.

Example:

```
public void setNumber(int number) {  // parameter declaration
    this.number = number;
}
```

Static Initializer

A declaration using the keyword static. A block of statements can be declared to be static, in which case it is executed when, and only when, the class is initialized.

Example:

```
public class Converter {
    public static int factor;  // static initializer
    static {   // static initializer
        factor = 5;
    }
}
```

super

A reference to the immediate super class.

Example:

```
public void execute() {
    // calling super class
    super.execute();  //  reference to super
    doMyOwn();
}
```

this

A reference to the current object.

Example:

```
public void startEngine(int type) {
    this.type = type;  //  reference to this
}
```

Variables

Variable declarations.

The engine uses the IsDecl property to distinguish between the actual declaration and references to the declaration.

Local Variable

A variable for which the scope is the current block.

Example:

```
     public void collect(String name) {
         boolean isGood = false;     // isGood is local to this
method
         isGood = getHasFunds();
     }
```

Field

Fields are the declarations of class or instance variables.

Example:

```
    public class Deck {
        public static int num_decks=0;           // static int field
         public final int NUM_CARDS = 52;          // int field
         private int[] cards = new int[NUM_CARDS]; //  array field
         public Deck() {
             num_decks++;
         }
     }
```

Expressions

Includes all types of expression nodes.

Assignment

Expressions utilizing assignment operators.

a=b

Direct assignment.

Assigns the value of b to a.

Example:

```
int x;
int y = 5;
x = y;     // a=b
```

a+=b

Add-equals assignment.

Assigns the result a+b to a.

Example:

```
int x=1;
x += 5;     // a+=b
```

a-=b

Subtract-equals assignment.

Assigns the result of a-b to a.

Example:

```
int x=5;
x -= 2;     // a-=b
```

a/=b

Divide-equals assignment.

Assigns the result of a/b to a.

Example:

```
float x = 10.0;
x /= 2.0;    // a/=b
```

a*=b

Multiply-equals assignment

Assigns the result of a*b to a.

Example:

```
int x = 2;
x *= 3;     // a*=b
```

a%=b

Mod-equals assignment.

Assign the remainder of a/b to a.

Example:

```
int x = 10;
x %= 7;     // a%=b
```

a&=b

Bitwise-And-equals assignment.

Assigns the result of a&b to a.

Example:

```
int x = 9;
x &= 5;     // a&=b
```

a^=b

Bitwise XOR assignment.

Assigns the result of a^b to a.

Example:

```
int x = 2;
x ^= 3;     // a^=b
```

a|=b

Bitwise OR assignment.

Assigns the result of a|b to b.

Example:

```
int x = 9;
x |= 8;     // a|=b
```

a<<=b

Left-shift-equals assignment.

Assigns the result of a<<b to a.

Example:

```
int x = 2;
x <<= 3;     // a<<=b
```

a>>=b

Right-shift-equals assignment.

Assigns the result of a>>b to a.

Example:

```
int x = 16;
x >>= 3;     // a>>=b
```

a>>>=b

Unsigned right-shift-equals assignment.

Assigns the result of a>>>b to a.

Example:

```
int x = 16;
a >>>= 3;     // a>>>=b
```

--a

Pre-decrement operator.

Decrement a before being used in expression.

Example:

```
int x = 5;
--x;         // --a
```

++a

Pre-increment operator.

Increment a before being used in expression.

Example:

```
int x = 5;
++x;         // ++x
```

a--

Post-decrement operator.

Decrement a after being used in expression.

Example:

```
int x = 5;
x--;           // a--
```

a++

Post-increment operator.

Increment a after being used in expression.

Example:

```
int x = 5;
x++;          // a++
```

Bitwise

All nodes using Java bitwise non-assignment operators.

~a

Bitwise negate operator.

Example:

```
int x = 7;
int y = ~x;     // ~a
```

a|b

Bitwise or operator.

Example:

```
int x = 3;
int y = 5;
int z = x|y;     // a|b
```

a&b

Bitwise and operator.

Example:

```
int x = 16;
int y = 9;
int z = x&y;     // a&b
```

a^b

Bitwise XOR operator.

Example:

```
int x = 16;
int y = 17;
int z = x^y;     // a^b
```

Comparison

All nodes using Java comparison operators.

a==b

Logical test for equality.

Example:

```
boolean method(int x, int y) {
    return x == y;              // a==b
}
```

a!=b

Logical test for inequality.

Example:

```
boolean method(int x, int y) {
    return x != y;              // a!=b
}
```

a<b

Logical test for less-than.

Example:

```
boolean method(int x) {
    return x > 5;              // a>b
```

```
}
```

a<=b

Logical test for less-than or equal-to.

Example:

```
boolean method(int x) {
    return x <= 5;              // a<=b
}
```

a>b

Logical test for greater-than.

Example:

```
boolean method(int x) {
    return x > 5;              // a>b
}
```

a>=b

Logical test for greater-than or equal-to.

Example:

```
boolean method(int x) {
    return x >= 5;              // a>=b
}
```

Logical

Nodes using logical operators.

!a

Logical negation.

Example:

```
boolean method(boolean x) {
    return !x;              // !a
}
```

a&&b

Logical 'and'.

Example:

```
boolean x = true;
boolean y = false;
boolean z = x && y;     // a&&b
```

a||b

Logical 'or'.

Example:

```
boolean x = true;
boolean y = false;
boolean z = x || y;     // a||b
```

Miscellaneous

Other operators not covered previously.

a.b

Dot operator.

Used to specify methods and fields of a particular class or object.

Example:

```
Object o;
String s = o.toString();     // dot operator
```

a[b]

Array Notation.

Specifies element with index = b of array a.

Example:

```
String[] args = new String[5];
args[3] = "sample string";     // Array notation.
```

instanceof

Tests whether or not an object is an instance of a particular class.

Example:

```
if (apple instanceof fruit) {   // Use of instanceof
...
}
```

a(b)

Method invocation.

Example:

```
class Foo {
    Foo() {
        setNum(3);              // a(b)
    }
    public void int setNum(int n) {
        num = n;
    }
    private int num;
}
```

a?b:c

Ternary operator.

Example:

```
int method(boolean ok) {
    return ok ? 0 : -1;   // a?b:c
}
```

Cast

Conversion of one data type into another.

Example:

```
char a = 'a';
int b = (int) a;    // Cast
```

new

Object instantiation.

Example:

```
public class AnyClass {
...
}
AnyClass aClass = new AnyClass();    // Object instantiation
```

Numerical

Nodes involving numerical operations.

+a

Represents the positive of a number.

The value of a remains unchanged.

Example:

```
int x = -6;
int y = +x;    // +a
```

-a

Represents the negative of a number.

Example:

```
int x = -6;
int y = -x;    // -a
```

a+b

The addition operator.

Example:

```
int x = 4;
int z = x + 3;     // a+b
```

a-b

The subtraction operator.

Example:

```
int x = 4;
int z = x - 4;     // a-b
```

a*b

The multiplication operator.

Example:

```
int x = 4;
int z = x * 3;     // a*b
```

a/b

The division operator.

Example:

```
int x = 4;
int z = x / 3;     // a/b
```

a%b

Mod operator.

Returns the remainder of a/b.

Example:

```
int x = 18;
int y = x%5;     // a%b
```

a<<b

Left bit-shift.

Example:

```
int x=2;
int y = x<<2;     // a<<b
```

a>>b

Right bit-shift.

Example:

```
int x=2;
int y = x>>2;     // right bit-shift
```

a>>>b

Unsigned right bit-shift.

Example:

```
int x = -1;
int y = y>>>1;    // a>>>b
```

Javadoc

Java comments.

Tags

Javadoc tags fir the comments (i.e., @author).

Packages

Nodes involving packages and imports.

import

The import statement.

Example:

```
import javax.swing.*;           // import statement
import javax.swing.border.*;    // import statement
```

package

The package statement.

Example:

```
package com.brijac.nn;    // package statement
```

Statements

Nodes involving various Java statements.

break

The 'break' statement.

Used to break out of a loop structure.

Example:

```
for (int t=1; t<5; t++) {
    if (t<2) break;     // break statement
}
```

case

The case statement.

Specifies a particular case of the switch statement.

Example:

```
int method(int i) {
  int ret = 0;
  switch (i) {
      case 2:          // case
          ret = 3;
          break;
      case 3:          // case
           break;
  }
  return ret;
}
```

continue

The continue statement.

Used to skip to the next iterator in a loop.

Example:

```
for (int t=0; t<5; ++t) {
    if ((t%2) != 0) continue;    // continue statement
    System.out.println(t);
}
```

do while

do while statement.

Example:

```
boolean done = false;
do {     // do while
} while (!done);
```

declaration statement

A statement involving the declaration of an object or primitive type.

Example:

```
int x = 7;                  // Declaration
Object o = new Object();    // Declaration
```

for

The 'for' loop.

Example:

```
for (int t = 1; t<5; t++) {    // for statement
...
}
```

if

The 'if' statement.

Example:

```
int x=5;
if (x==5) {    // The if statement
    System.out.println("x="+x);
}
```

return

The return statement.

Example:

```
int x = 7;
public void int getNumber() {
    return x;      // return statement
}
```

synchronized

The synchronized statement.

Locks object when in use so that it can be accessed by only one caller at a time.

Example:

```
public class Foo
{
  public static Thread get ()
  {
    final Object resource1 = "resource1";
    return new Thread () {
        public void run () {
           synchronized (resource1) {  // synchronized
           }
        }
    };
  }
}
```

switch

The switch statement.

Example:

```
    int t=2;
    switch(t) {        // switch
        case 1:
            break;
        case 2:
             break;
    }
```

throw

The throw statement.

Example:

```
    void method(Object t) {
        if (t == null) {
            throw new Exception();
        }
    }
```

try

The try statement.

Example:

```
try {    // try statement
} catch (Exception e) {
}
```

while

The while statement.

Example:

```
boolean here = true;
while (here) {     // while statement
}
```

Block

The code block

Example:

```
{   // A block
    // statements within this block
 }
```

Simple

A simple statement.

A singular code statement which does not fall in any of

the other statement categories.

Example:

```
for (int count=1; count<10; count++) {
    System.out.println(count);     // simple statement
}
```

Types

Nodes representing complex or primitive types.

Complex

Non-primitives.

Array

The array type.

Example:

```
String[] stringArray = new String[5];     // array
```

Class

The Java class type.

Example:

```
public class Foo {     // class
}
```

Interface

The Java interface type.

Example:

```
public interface Plug {     // interface
}
```

Primitive

Nodes representing primitive types.

boolean

The boolean primitive type.

Example:

```
boolean bool = true;     // boolean
```

char

The character primitive type.

Example:

```
char a = 'a';     // char
```

byte

The byte integral type.

Example:

```
byte b = 23;     // byte
```

short

The short integer.

Example:

```
short s = 4;     // short
```

int

The integer type.

Example:

```
int t = 5;     // int
```

long

The long integer.

Example:

```
long size = 84578734;     // long
```

float

The floating point type.

Example:

```
float width = 78.3e+12f;     // float
```

double

The double numeric type.

Example:

```
double length = 93.45e+301d;     // double
```

Javadoc

Tags

==========

Index

N

O

P

Q

R

S

T

U

V

X