Checkstyle

Checkstyle is a development tool to help programmers write Java code that adheres to a coding standard. It automates the process of checking Java code to spare humans of this boring (but important) task. This makes it ideal for projects that want to enforce a coding standard.

Checkstyle is highly configurable and can be made to support almost any coding standard. An example configuration file is supplied supporting the [Sun Code Conventions](http://java.sun.com/docs/codeconv/). As well, other sample configuration files are supplied for other well known conventions.

It would catch things like missing/improper javadoc, naming conventions, placement of braces and parentheses, whitespace, line length, etc.

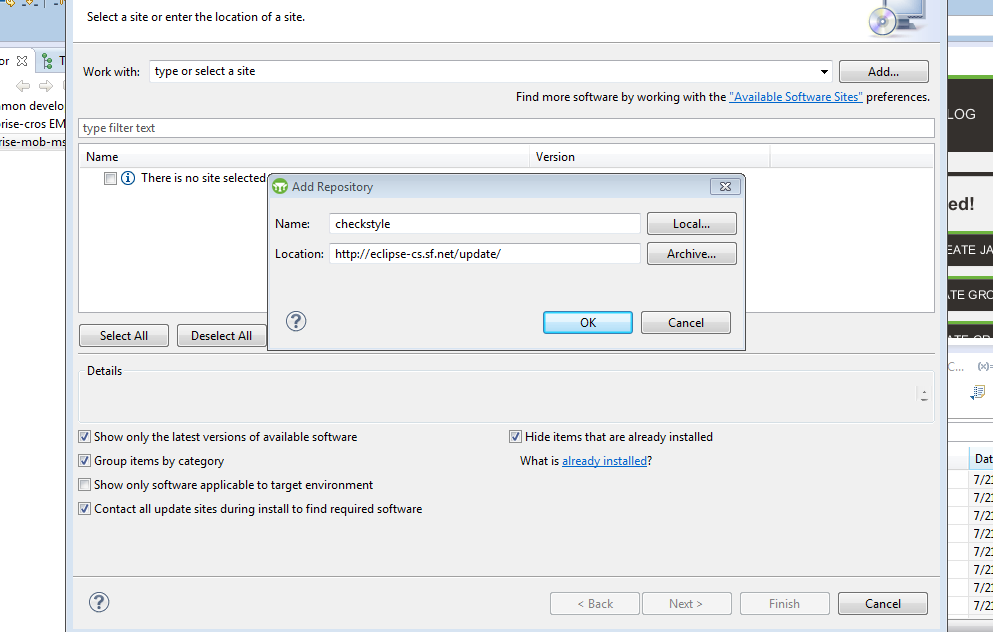
You can download latest version of checkstyle from the following link :

<http://checkstyle.sourceforge.net/>

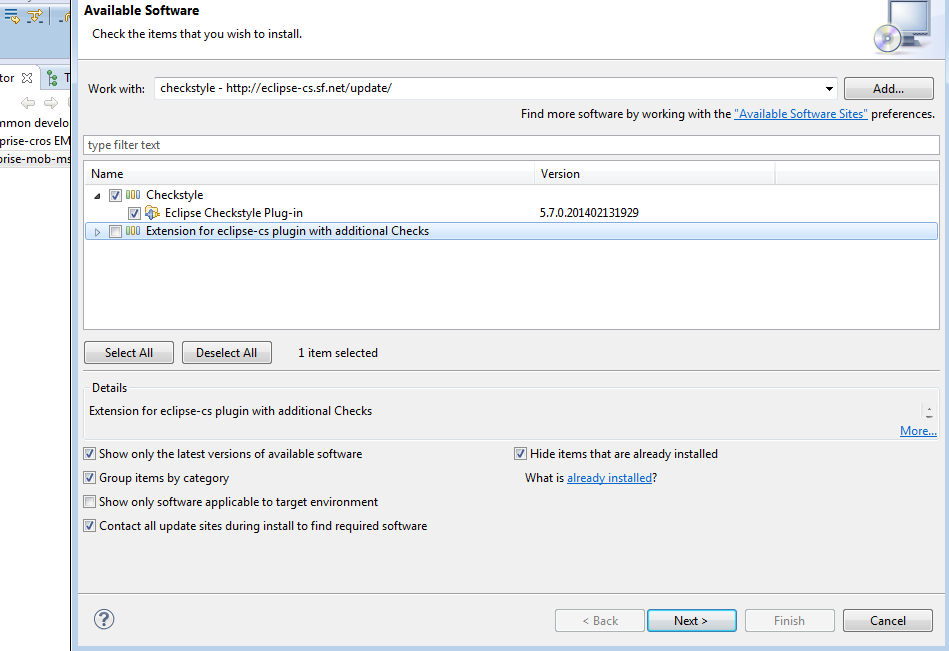
Checkstyle using eclipse plugin:

Installation steps:

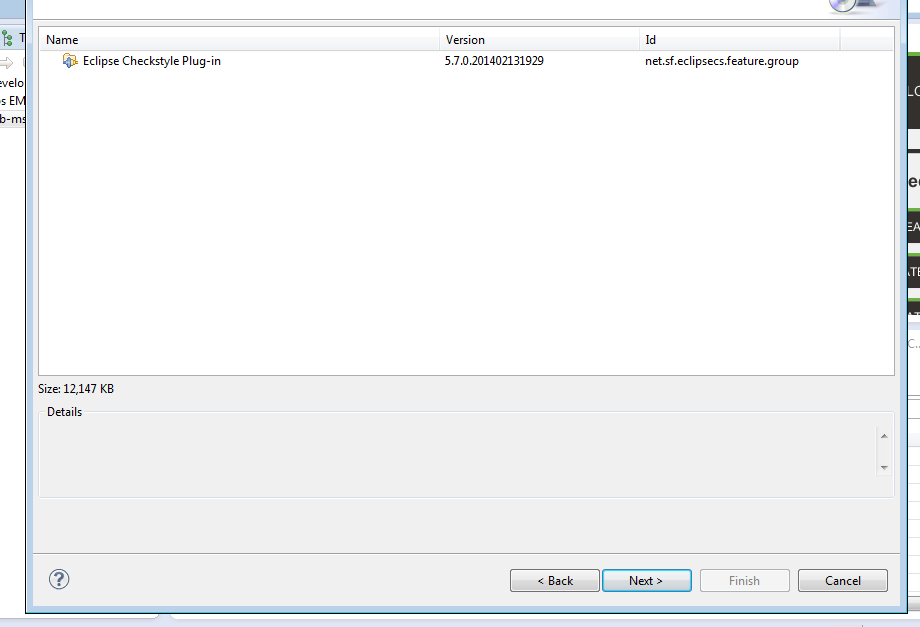
Step 1: go to help > install new software. Add a new URL <http://eclipse-cs.sf.net/update/> as shown below.



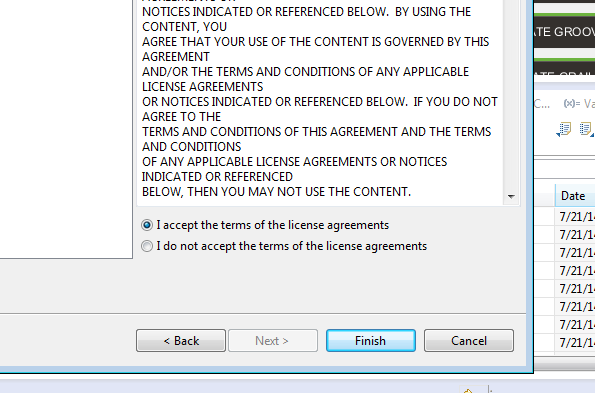
Step 2: select Eclipse checkstyle plugin and click next.



Step 3: again click next



Step 4: accept terms of license agreement and click finish.

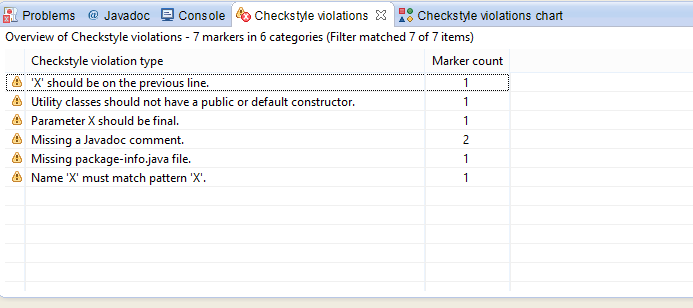


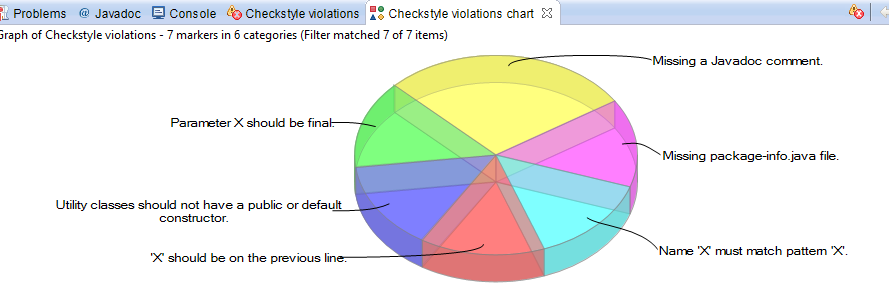
**Scan your project against default checkstyle rules:**

Step1: right click your project goto **checkstyle** and then select **check code with checkstyle.** It will validate your project against the provided rule set.

Step 2: Now you can see the result in different views like checkstyle violations and checkstyle violations chart.

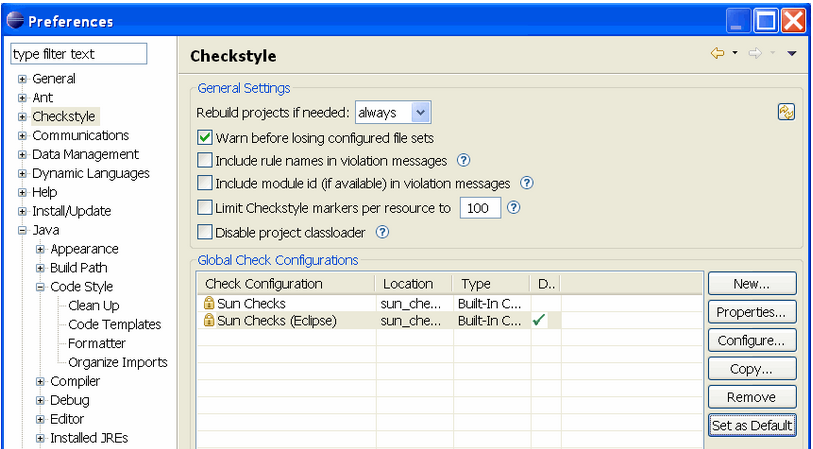
Step 3: you can clean those views by right clicking your project goto **checkstyle** and then select **clean checkstyle violations.**

****

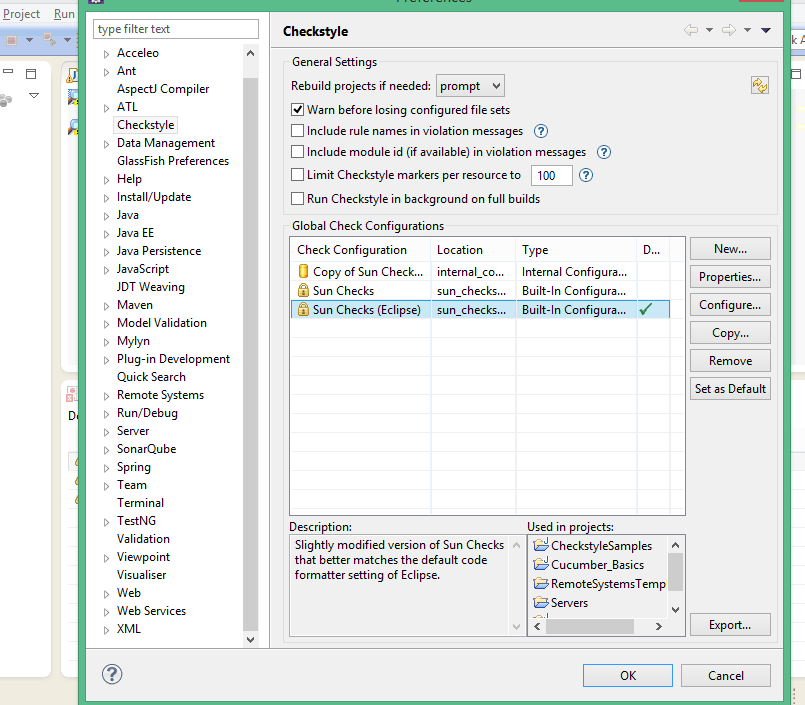
****

**Enable or disable rules:**

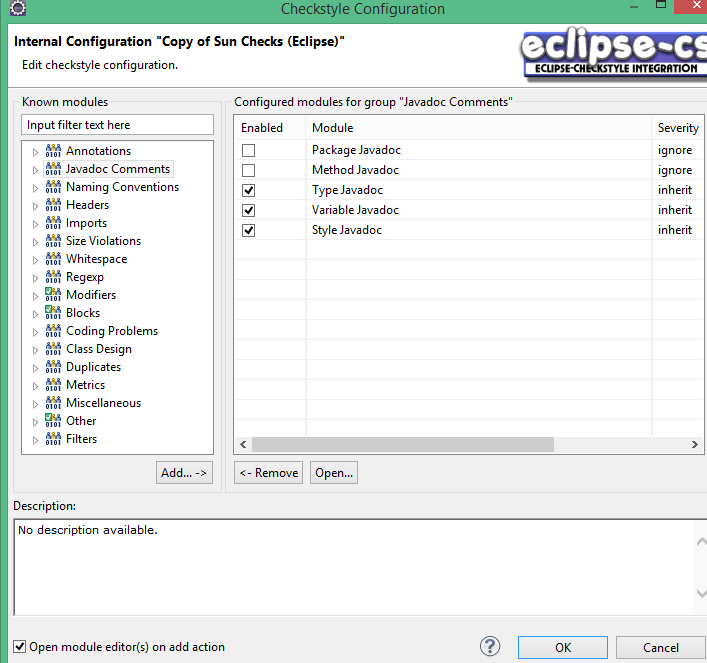
**Goto** *Window* → *Preferences* → *Checkstyle*. Then you will see as shown below.

****

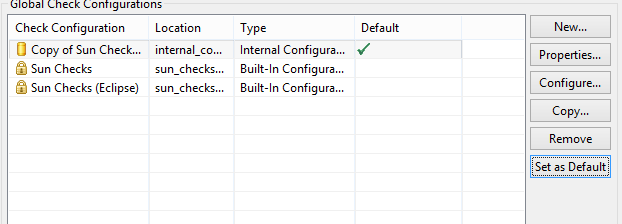
You can turn off certain checks. To customize first make a copy of the existing profile by clicking the copy button. Now you will find a copy of existing profile gets created as shown below.



Now you can edit the new profile by selecting the profile and clicking **configure** button.Now you will find there is checkbox to the left of every rule, through which you can enable or disable different rules as shown below.



When you are done with editing click OK button and set your new new profile as default by clicking “Set as default” button. You will find a green check mark to the right of your selected profile which indicates your current active profile.



Now if you validate your project with checkstyle, it will be validated against the current active rules.

**Automated code reviews with Checkstyle**

Code reviews are essential to code quality, but no team wants to review tens of thousands of lines of code, or should have to. In this two-part article, ShriKant Vashishtha and Abhishek Gupta show you how to overcome the challenges associated with code reviews by automating them. Find out why Checkstyle is one of the most popular tools used for code review automation, then learn how to quickly enhance its built-in rules with custom ones just for your project. Level: Intermediate

This white paper demonstrates why businesses choose Box to unify their storage and collaboration and

If you've worked on a large-scale project you know first-hand the value of an [automated code review](http://www.javaworld.com/javaworld/jw-11-2005/jw-1114-j2ee.html). Really big projects require the input of hundreds of programmers, often geographically dispersed and with great differences in skill level. Code written by engineers interacts with code written by novices in such projects -- haiku interspersed with high-school poetry.

In many cases, a QA team is assigned to review this code manually, based on coding standards used as guidelines for development. Manually reviewing millions of lines of code is a tedious job, though; ultimately as exhausting as it must be exhaustive.

Smart teams don't do code reviews manually: instead they rely on source code analyzers like [Checkstyle](http://checkstyle.sourceforge.net/), [PMD](http://pmd.sourceforge.net/), and [JTest](http://www.parasoft.com/jsp/products/home.jsp?product=Jtest). Such tools come with readymade rules that help in maintaining code standards. These rules are a good starting point, but they don't account for project-specific requirements. The trick to a successful automated code review is to combine the built-in rules with custom ones. The more refined your rules, the more truly automated your code review becomes.

Given the benefits of automated code review, you might expect more people to do it. In fact, many developers want to implement project-specific custom rules like those available with Checkstyle. To the uninitiated, custom rule creation seems difficult and time-consuming. There's very little documentation on the internals of code review tools, and very few tutorials show how to create custom checks.

In this first article in JavaWorld's two-part introduction to Checkstyle, we'll remedy that situation, making the task of writing custom Checkstyle rules so simple that any Java developer can potentially do it in a day. After reading this article, you will be able to write your own custom Checkstyle rules without the help of specialized skills. In Part 2, we'll show you how to be more proactive about code quality, by [stopping faulty code before it enters your code base](http://www.javaworld.com/javaworld/jw-11-2008/jw-11-checkstyle2.html).

## Checkstyle and Java grammar

Checkstyle is a free and open source [development tool](http://www.javaworld.com/podcasts/jtech/2008/022808jtech.html) that helps ensure that your Java code conforms to the coding conventions you've established. It automates the boring but crucial task of checking Java code. Checkstyle is often used as an [Eclipse plugin](ttp://sourceforge.net/projects/eclipse-cs/), and also as part of a project build to create a report of coding-standard violations. It can be used in conjunction with build tools such as [Ant](http://www.javaworld.com/javaworld/jw-11-2005/jw-1107-build.html) or [Maven](http://www.javaworld.com/javaworld/jw-12-2005/jw-1205-maven.html). Checkstyle provides many readymade standard coding rules, which are very useful. However, this article focuses on creating custom rules that are more useful in enterprise development.

Before you write any custom rules for Java files, you need to consider the grammar used to write those Java files. Whenever you think about Java classes, a certain structure comes to mind. A Java class begins with a package definition, followed by import statements. In the object block (for a class or interface) you will find instance variables, a constructor, and methods. You could compare this to an XML tree structure. When you want to read an XML file, you use a parser. You the same thing with Checkstyle, but for Java files. Checkstyle uses the [ANTLR Parser](http://www.javaworld.com/javaworld/jw-08-2008/jw-08-dsls-in-java-3.html). Figure 1 illustrates the tree structure you get when the ANTLR Parser takes on a Java file.

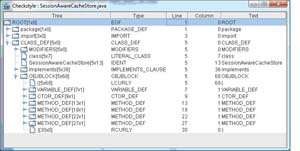
[[](http://www.javaworld.com/javaworld/jw-11-2008/images/checkstyle1-fig1.jpg)](http://www.javaworld.com/javaworld/jw-11-2008/images/checkstyle1-fig1.jpg" \t "_blank)

Figure 1. Tree structure of a Java file (click to enlarge)

You'll find no surprises in the structure shown in Figure 1. Continuing with the metaphor of an XML structure, the Type column in Figure 1 corresponds to XML tags, the Text column corresponds to the value of a tag, and the Line and Column columns correspond to tag attributes.

Checkstyle provides a Java Swing GUI tool that will let you view the tree structure for your Java files. You can invoke this tool with the following command:

java -classpath checkstyle-all-<version>.jar com.puppycrawl.tools.checkstyle.gui.Main <JavaFileToParse>

To produce Figure 1, we used SessionAwareCacheStore.java on the command line in place of <JavaFileToParse>. This class, among the many others discussed in this article, is included in the article source, [checkstyle-src.zip](http://images.techhive.com/downloads/idge/imported/article/jvw/2008/11/checkstyle-src.zip). This package provides the code for all the Checkstyle rules, or checks used in this article. It also contains some very useful utility classes that simplify the task of writing Checkstyle checks, along with a readme.txt file that provides details on how to build, configure, and use these custom checks.

The Java tree structure shown by the GUI tool forms the basis for the creation of Checkstyle checks. The tree structure helps define the test cases for creating them.

## How does Checkstyle work?

When you say that you want to write a custom Checkstyle rule or check, you're essentially saying that you want to write a class that extends the Check class. Checkstyle is implemented in terms of modules of checks. Modules can contain other modules and hence form a tree structure, as you can see in Listing 1.

#### Listing 1. File containing the list of modules in a custom Check configuration file

<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE module PUBLIC "-//Puppy Crawl//DTD Check Configuration 1.2//EN" "http://www.puppycrawl.com/dtds/configuration\_1\_2.dtd">

<module name="Checker">

<property name="severity" value="error" />

<module name="TreeWalker">

<property name="severity" value="error" />

<module

name="com.abc.checkstyle.check.IllegalMethodCallInForCheck">

<property name="severity" value="error" />

<property name="methodNames" value="length,size" />

</module>

</module>

</module>

You Might Also Like

Custom rules can be used for the purposes of code analysis, as well as review automation. For instance, if you suspect a memory leak in an application, one of the first steps you should take is to a look at the [Java collection classes](http://www.javaworld.com/javaworld/jw-02-2000/jw-02-performance.html), which often do not discard objects that are no longer in use. As a rule, you may want to look at static collections. The problem becomes a bit more complex, however, when a collection as a variable is not static, but the class that contains it is a static data member for another class. In those cases, simple grep-like tools may not do the trick. Rather than analyze thousands of classes, try creating a custom Checkstyle rule that will pick up all the instances of collection instance variables in different classes. This rule could also pick up all the data members in and out of the collection. Once you know how to write custom rules, you find that they are useful for more than just code reviews.

The Checkstyle kernel interacts with modules that implement the FileSetCheck interface. Checkstyle provides some FileSetCheck implementations by default. One of them is TreeWalker. TreeWalker walks through all modules (classes) that derive from the Check class. To write a custom rule, you need to extend the Check class and plug it into the Checkstyle configuration file.

In Listing 2, the class com.abc.checkstyle.check.IllegalMethodCallInForCheck is a custom Checkstyle rule that extends from the Check class.

TreeWalker is based on the fundamentals of the Visitor pattern. It walks through all classes that extend from the Check class. However, as a custom rule developer, you can specify the event that should prompt TreeWalker to visit a particular extension of the Check class.

## Your first custom check

Imagine that you want to implement a rule, or check that gives you a warning when you put java.lang.Exception in a throws clause. In a Java class tree, you can specify that whenever the TreeWalker walks through a throws clause (the LITERAL\_THROWS token in Figure 2), it should visit your Check class.

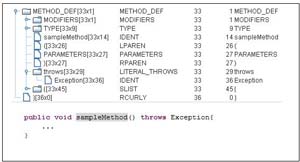
[[](http://www.javaworld.com/javaworld/jw-11-2008/images/checkstyle1-fig2_thumb.jpg)](http://www.javaworld.com/javaworld/jw-11-2008/images/checkstyle1-fig2_thumb.jpg" \t "_blank)

Figure 2. Sample throws clause (click to enlarge)

To achieve this, you need to specify a token (LITERAL\_THROWS) in the getDefaultTokens() method of your custom Check implementation. This token tells the TreeWalker to call the IllegalExceptionThrowsCheck class, as shown in Listing 2.

#### Listing 2. getDefaultTokens() method definition in IllegalExceptionThrowsCheck

...

public final class IllegalExceptionThrowsCheck extends Check {

@Override

public int[] getDefaultTokens() {

return new int[] { TokenTypes.LITERAL\_THROWS };

}

...

...

}

getDefaultTokens() is an abstract method of the Check class; you need to implement it in every custom Check class. TreeWalker uses the getDefaultTokens() method to determine the tokens on which TreeWalker should call the check. In the current scenario, you've specified the LITERAL\_THROWS token as the trigger to call IllegalExceptionThrowsCheck.

As you might have noticed, getDefaultTokens() returns multiple tokens. It indicates that Check will be called for all tokens you return from this method. For example, a Check class with the getDefaultTokens() implementation in Listing 3 will be called whenever TreeWalker encounters the for, while, or do-while tokens.

#### Listing 3. getDefaultTokens() method definition with multiple tokens returned

public int[] getDefaultTokens() {

return new int[] { TokenTypes.FOR\_CONDITION, TokenTypes.LITERAL\_WHILE,

TokenTypes.DO\_WHILE };

}

Coming back to the IllegalExceptionThrowsCheck example, whenever TreeWalker walks through the LITERAL\_THROWS node in the tree, the visitor method visitToken() (overridden from the Check class) on IllegalExceptionThrowsCheck will be called. The DetailAST parameter contains the information about the visited token (the throws token, in this case) and becomes the input for writing a check.

Before you write the implementation of the visitToken() method on IllegalExceptionThrowsCheck, take a look at the throws node in Figure 2. The node type is LITERAL\_THROWS, which is why you need to specify LITERAL\_THROWS in the getDefaultTokens() method. This node has sub-nodes under it, which in turn specify all exceptions under the throws clause.

The rest should be pretty clear. You need to find child nodes under the LITERAL\_THROWS node that match the test condition -- java.lang.Exception, in this case. That's all that needs to happen in the visitToken() method, which is shown in Listing 4.

#### Listing 4. visitToken() method implementation in IllegalExceptionThrowsCheck

...

public final class IllegalExceptionThrowsCheck extends Check {

...

...

public void visitToken(DetailAST aDetailAST) {

DetailAST identDetailAST = aDetailAST.findFirstToken(TokenTypes.IDENT);

if ("Exception".equals(identDetailAST.getText())) {

log(aDetailAST, "Illegal Throws Clause -> "

+ identDetailAST.getText());

}

}

}

Note that, in the visitToken() method, aDetailAST is the object for the identified token (LITERAL\_THROWS, for this check) containing all the required details for the processing logic. Among other things, DetailAST provides methods to get parent, child, and neighboring nodes. It's nothing but a tree-leaf structure, or an implementation of the [Composite pattern](http://www.javaworld.com/javaworld/jw-09-2002/jw-0913-designpatterns.html). The manner in which Checkstyle displays the error is determined by the log() method.

The current example just deals with a single token returned by getDefaultTokens(). At this point, you may be wondering: if this method returns multiple tokens, how can you handle the traversal of those tokens in visitToken()? Just as DetailAST contains all the information about the token, it also contains information about the type of node. Based on the type of node you're dealing with, you can have different code execution paths in the visitToken() method.

### Configuring your check

The name of the exception was hardcoded in Listing 4. Wouldn't it be nice to keep the exception in a configuration file? That's certainly possible in Checkstyle; you can define the external property inside the XML configuration file for this check, as shown in Listing 5.

#### Listing 5. External property illegalExceptionsInThrows in IllegalExceptionThrowsCheck configuration

<module name="IllegalExceptionThrowsCheck">

<property name="severity" value="error" />

<property name="illegalExceptionsInThrows" value="Exception,java.lang.Exception" />

</module>

You need a corresponding setter method for this property as well. Listing 6 shows the related changes that need to be made to the Check class.

#### Listing 6. Using an external property in IllegalExceptionThrowsCheck

...

public final class IllegalExceptionThrowsCheck extends Check {

List<String> illegalExceptionsInThrows

public void visitToken(DetailAST aDetailAST) {

...

if (illegalExceptionsInThrows.contains(identDetailAST.getText())) {

...

}

}

public void setIllegalExceptionThrows(String illegalExceptionThrowsStr) {

this. illegalExceptionsInThrows = Arrays.asList(illegalExceptionThrowsStr

.split(","));

}

}

Congratulations! You've created your first custom Check.

### Testing your check

To see how this would work in practice, you need to execute your Check class with a test class. By putting the Check class in a Checkstyle configuration file, as in Listing 7, you can tell Checkstyle to execute the newly created IllegalExceptionThrowsCheck.

#### Listing 7. custom\_check.xml

<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE module PUBLIC "-//Puppy Crawl//DTD Check Configuration 1.2//EN" "http://www.puppycrawl.com/dtds/configuration\_1\_2.dtd">

<module name="Checker">

<property name="severity" value="error" />

<module name="TreeWalker">

<property name="severity" value="error" />

<module name="IllegalExceptionThrowsCheck">

<property name="illegalExceptionsInThrows" value="Exception,java.lang.Exception" />

</module>

</module>

</module>

Now enter the following at the command line:

java -classpath checkstyle-all-4.4.jar com.puppycrawl.tools.checkstyle.Main -c custom\_check.xml ExceptionTestCase.java

You'll see the following output:

Starting audit...

C:\workspace\others\CheckSample\ExceptionTestCase.java:33:37: Illegal Throws Clause -> Exception

Audit done.

Ta da! You can see that the throws clause that violates the policy you've laid down has been detected.

## Elements of a custom check

As you might have noticed, you don't have to be an expert in ANTLR to be able to write your own custom checks. Using the GUI tool provided by Checkstyle, you can understand test conditions better and see how different nodes are interacting in a Java tree structure. And that becomes the basis for implementing a check.

You may find the terminology used in the tree a bit confusing. Table 1 provides you with a list of the nodes of a Java tree structure and their corresponding tokens to help you to write a check.

#### Table 1. Token types for different Java class constructs

|  |  |  |
| --- | --- | --- |
| **Java structure** | **Corresponding constant (TokenTypes.<constant name>)** | **Example** |
| Class definition | CLASS\_DEF | public class Sample{ |
| Constructor definition | CTOR\_DEF | public Sample(){}  parses as:  +--CTOR\_DEF  |  +--MODIFIERS  |  +--LITERAL\_PUBLIC (public)  +--IDENT (Sample)  ... |
| Method definition | METHOD\_DEF | public static int square(int x){  return x\*x;  }  Parses as:  +--METHOD\_DEF  |  +--MODIFIERS  |  +--LITERAL\_PUBLIC (public)  +--LITERAL\_STATIC (static)  ... |
| Variable definition | VARIABLE\_DEF | String sample;  Parses as:  +--VARIABLE\_DEF  |  +--MODIFIERS  +--TYPE  |  +--IDENT (String)  +--IDENT(sample)  +--SEMI(;) |
| throws clause | LITERAL\_THROWS | throws |
| Method call | METHOD\_CALL | Thread.sleep()  Parses as:  +--METHOD\_CALL (()  |  +--DOT (.)  |  +--IDENT (Thread)  +--IDENT (sleep)  +--ELIST  +--RPAREN ()) |
| String | STRING\_LITERAL | "sample" |
| Catch | LITERAL\_CATCH | The catch keyword |
| for, while, do-while | FOR\_CONDITION, LITERAL\_WHILE, DO\_WHILE | for(int i=0;i<list.size();i++)  The for condition (i<list.size()) parses to:    +--FOR\_CONDITION (()  |  +--EXPR  |  +--LT (<)  |  +--IDENT (i)  +--METHOD\_CALL (()  |  +--DOT (.)  |  +--ELIST  +--RPARAM ()) |
| Object block (Children of class, interface, etc). | OBJBLOCK | public interface InterfaceSample{  void method(String [] args);  }  Parses to:  +--EOF (ROOT)  |  +--CLASS\_DEF  |  +--MODIFIERS  |  +LITERAL\_PUBLIC (public)  +--LITERAL\_INTERFACE (interface)  +--IDENT (InterfaceSample)  +--OBJBLOCK  |  +--METHOD\_DEF |
| Parameter declaration | PARAMETER\_DEF | public void methodA(String [] args) |

## Custom checks: More complex rules

So far you have written a very basic check for identifying an unwanted throws clause. Now it's time to think about more complex rules.

When you call a method on a null object without checking null, it definitely throws an exception. With low test coverage in your code, you may be in peril of discovering this when the code is actually in production -- i.e., too late. You can create a check that warns you when you call a method on an uninitialized object. Listing 8 includes some of the test cases.

#### Listing 8. Test conditions for null check custom rule

//Test Case 1

String x = null; //Initialized to null

x=new String(); //Initialized now

x.length(); //A valid operation

//Test Case 2

String name = "Prerana"; //Initialized

//Test Case 3

String name2 = x; //Could be a potential problem if x were null. Not a case here though.

//Test Case 4

String nullString= null; //Not initialized

nullString.length(); //Invalid operation

//Test Case 5

List list1; //Not initialized

List list2 = list1; // Uninitialized. Assigned list1 is not initialized

list2.getSize(); // Invalid operation

//Test Case 6

list2 = new ArrayList();// Initialized

//Test Case 7

sample = sampleMethod(); // Potentially null assignment

sample.length(); // No Null check. Potential problem

Now take a look at some of these test cases in more detail. The first thing you'll notice is that variables are created and initialized within a scope. This check is only focused on determining if a variable has been initialized in the current scope or in deeper nested scopes, as shown in Listing 9.

#### Listing 9. Variable initialization in current and nested scope

{//Scope 1 begin

String x = null;

{//Another scope 2 begin

Map map1 = null; //Map declared in scope 2

x = new String(); //x initialized in this scope not at a place where it got declared

...

map1 = new HashMap();

} //map1 unreferenced. Available for garbage collection. Scope 2 ends

} //variable x unreferenced. Available for garbage collection. Scope 1 ends.

Take another look at Listing 8, and note that test cases 4, 5, and 7 have variables uninitialized. The rest of the cases are for initialized variables. Variables can be initialized somewhere in the siblings (downward in the method implementation) of the VARIABLE\_DEF tokens, as you can see in the Checkstyle GUI in Figure 3. They can also be initialized somewhere in the nested scopes, as shown in Listing 9.

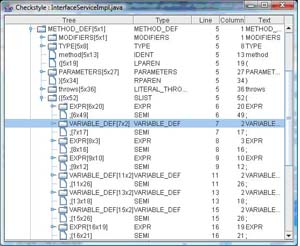
[[](http://www.javaworld.com/javaworld/jw-11-2008/images/checkstyle1-fig3.jpg)](http://www.javaworld.com/javaworld/jw-11-2008/images/checkstyle1-fig3.jpg" \t "_blank)

Figure 3. Downward siblings of a variable definition (click to enlarge)

Note that some of the objects, such as a String, can be initialized using the new operator, as well as with the String literal. In case 2 of Listing 8, Prerana is used as the literal.

### Implementing the check

Now it's time to implement this check. It should be pretty clear that you need to focus on variable initialization inside a method, so you're looking for the token: METHOD\_DEF. Listing 10 contains the implementation of the getDefaultTokens() method.

#### Listing 10. getDefaultTokens()

@Override

public int[] getDefaultTokens() {

return new int[] { TokenTypes.METHOD\_DEF };

}

Next, you need to tackle the visitToken() method. First of all, you need to get the list of all nodes of the variable definitions in the method. If you check the representation of the variable definitions in Figure 3, you'll see that that's all that's happening here. Method definition entails many things other than method implementation itself -- parameter declaration, for instance -- but you should just focus on method implementation in this check. As you can see in Figure 3, you need to get the first node of the SLIST token to get hold of the method definition; from that, you can get the list of VARIABLE\_DEF nodes. This is illustrated in Listing 11.

#### Listing 11. Getting all instances of variable definition in method scope

public void visitToken(DetailAST aast) {

super.visitToken(aast);

DetailAST slist = aast.findFirstToken(TokenTypes.SLIST);

List<DetailAST> variables = DetailASTUtil

.getDetailASTsForTypeInBranch(slist, TokenTypes.VARIABLE\_DEF);

Listing 11 uses DetailASTUtil's getDetailASTsForTypeInBranch() method to get all nodes in the branch for a specific token type (VARIABLE\_DEF, in this case). DetailASTUtil provides provides much-needed utility methods for DetailAST; take a closer look at the sample code package for details.

Next, based on the analysis performed for uninitialized variables in Listing 8, you can filter out initialized variables from the variables list using the isVariableInitialized() method of the custom-built JavaClassStructureUtil class. You're left with a list of uninitialized variables, called uninitializedVars.

Now there's one more thing left to check for: whether or not a method has been called on a variable before it's initialized. For that, you must iterate over the list of uninitialized variables. For each uninitialized variable, you search the lists of nodes among its siblings, where either the uninitialized variable has been assigned (TokenTypes.ASSIGN) or a method has been called on it (TokenTypes.DOT). This is shown in Listing 12.

#### Listing 12. Determining whether a method has been called on a variable before it is initialized

for (DetailAST uninitializedVar : uninitializedVars) {

...

List<DetailAST> siblingsBelow = DetailASTUtil.getAllSiblingsBelow(uninitializedVar);

for (DetailAST siblingBelow : siblingsBelow) {

getListOfMethodCallsOnVariable(uninitializedVarName,

siblingBelow, methodCallsFullIdentsList);

getListOfVariableAssignment(uninitializedVarName

, siblingBelow, assignmentFullIdentsList);

}

With the getListOfMethodCallsOnVariable() method, you get a list of nodes (DOT tokens) where the method has been called on the variable; with the getListOfVariableAssignment() method, you get a list of nodes (ASSIGN tokens) where the variable has been assigned to something. You may want to take a look at the source code of this check to get the details of getListOfMethodCallsOnVariable() and getListOfVariableAssignment().

The lists you receive from these methods contain a FullIdent object for each node. FullIdent contains information about the location (line number, column) of the node. Based on the locations of ASSIGN (potential object creation) or DOT (method call) nodes, you can decide whether a method has been called on an uninitialized variable before variable creation. If that's the case, you log an error message for that variable.