JavaChecker Guide.

Version 2.5
Ruslan Shevchenko <Ruslan@Shevchenko.Kiev.UA>
Grad-Soft Ltd (http://www.gradsoft.ua), Kiev, Ukraine.

Table of Contents

Introduction

JavaChecker is a static analyzer tool for java language, based on TermWare technology. It's provide not only set of software checks, but framework for implementing own ones on top of semantics model of Java project.

Installation

1. Download binary or source installations from Files section of javachecker project on java.net website (https://javachecker.dev.java.net/) or from gradsoft public repository (http://datacenter.gradsoft.ua/public.repository/ua.gradsoft/javachecker/)

For binary installation file must named **JavaCheckerBinaryInstaller-*(version)*.jar**, for source – **JavaCheckerSourceInstaller.*(version)*-jar**, where (version) is 3-digit version identifier, separated by dots.

2. run jar archive with command, for example:

```
java -jar JavaCheckerSourceInstaller-2.5.1.jar
```

Invoking

Command line.

Invoking from command line is simple: you can use JavaChecker2.bat (or sh) from *bin* subdirectory of installation package with source directory as argument and may be some additional output. --help option show next:

```
Usage: JavaChecker [options] directory
where options must be one from:
  --prefs fname              read configuration from preferences file fname.
  --showFiles                during check, print names of analyzed files.
  --help                     output this help message.
  --output fname             write report to file fname
  --output-format (text|html|xml)   set report format.
  --enable check-name        enable checking for check-name.
  --disable check-name       disable checking for check-name.
  --explicit-enabled-only    run only checks, that enabled in command line.
  --config name  value       set configuration item of name to value.
  --include dirname          set directory, where situated source files, from
which processed sources are depend.
  --attrdir dirname          set directory, where tree of external source
attributes is situated.
  --statistic-detail (file|directory|all)
                             output statistics for each file, for each directory
                             of for all sources at one. (default - all)
  --debug                    put to stderr a lot of debug output
  --dump                     dump to stdout AST and semantics models of parsed
files.
  --q                        minimize output to stdout
```

ANT task

JavaChecker can be integrated with apacha ant build tool. You can just add next task definition to you build.xml file:

```
 <taskdef name="javachecker"
classname="ua.gradsoft.javachecker.ant.JavaCheckerTask">

     <classpath>

         <fileset dir="${jchhome}/lib"  includes="**/*.jar" />

     </classpath>

</taskdef>
```

where ${jchhome} is a property which point to directory where JavaChecker is installed.

From version 2.5.1 it is possible to define javachecker namespace:

```
 < xmlns:javachecker="antlib:ua.gradsoft.javachecker>"
```

and load resource libraries via next target:

```
 <target name="declare-javachecker"  depends="download-build-libs">
   <path id="javachecker.path">
    <fileset dir="${jchhome}/lib" includes="**/*.jar" />
   </path>
   <taskdef resource="ua/gradsoft/phpjao/antlib.xml"
            uri="antlib:ua.gradsoft.phpjao"
            classpathref="phpjao.lib.path"/>
 </target>
```

Then you can use ant task javachecker with following attributes:

- jchhome - required, must be set to directory, where javachecker is installed.
- input – required if inputs tag are absent, must be set to directory, where located java files, which you want to check.
- include – set to directory, where located additional source files.
- attrdirs -- set to directory, where located external source attributes.
- output - optional, must be set to filename, where report will be printed. Otherwise, report will be printed to stdout.
- prefs - optional, must be set to xml prefernces file with JavaChecker configuration.
- showfiles - optional, analog of --showFiles option
- q - optional, analog of -q option.
- outputFormat -- optional, set format of javachecker report. Can be “text” or “html” or “xml”
- fork – optional, run javachecker in separate process.
- failOnError – optional, fail if javachecker can't process files. (by default – true).
- explicitEnabledOnly -- optional, run only checks, that explicit enabled inside task. (by default - false)
- StatisticDetails -- optional, set level of detalization for statistics (one of 'file', 'dir', 'all')

And following internal tags:

- input, attributes are:
  - dir   -- name of directory with java files to process.
- include, attributes are:
  - dir -- name of directory with java files, to look for if necessory.
- enable -- enable specific checks, attributes are:
  - check   -- must be set to name of check to enable.
- disable – disable specifics checks, attributes are:
  - check -- name of check to disable.
- config – set confiuration value, attributes are:
  - name -- name of value to set.
  - value -- value to set.
- classpath -- path like structure, which point to jars, from which checked application is depend.
- jvmarg -- pass jvam argument for java process. Has effect only if fork=true.

Example:

```
 <target name="check" depend="build" >
   <javachecker:check jchhome="${jchhome}"
        input="src" output="jchreport.txt" prefs="etc/jchprefs.xml" />
 </target>
```

will process all files in src.

Next task definition:

```
<target name=”check”  depend=”build” >
 <javachecker:check jchhome=”${jchhome}”  fork=”true” >
      <jvmarg value=”-Xmx512M” />
     <input  dir=”src” />
     <input   dir=”src1” />
     <include dir=”src-ext” />
      <classpath>
          <fileset dir=”lib”>
                <include name=”*.jar” />
          </fileset>
      </classpath>
      <disable  check=”NamePatterns” />
 </javachecker>
</target>
```

will process all files in src and src1, disable checking of NamePatterns and if during processing of src or src1, we will need information from other classes, JavaChecker will look at first to src-ext for sources, then will try to load needed classes from lib/*.jar

Also javachecker will run in separate process and option -Xmx512M will be passed to child JVM.

And next task:

```
<target name="check-loops" depend="build" >
  <javachecker:check jchhome="${jchhome}"
      input="src" output="jchreport.txt"
      explicitEnabledOnly=”true”
      >
      <enable check=”ForAroundSizeConvertable” />
  </javachecker>
</target>
```

will search in src all cases, where

Configuration

Configuration of JavaChecker consists from:

- definitions of checkers and counters;
- config items
- attributes of source objects.

Checkes definitions:

Checkers are defined in files with names like etc/checkers_<something>.def Typical definition looks like:

define(name, category, message, type, ruleset (or name of class), enabled)

Detailed description of check structure is described in cpecial chapter, what is needed for configuration: that

- we can use **name** when enable or disable check by ant task element or command-line argument;
- last element (**enabled**) can be set to true or false, we can modify one to enable or disable processing of given check.

Config items:

Config items (such as patterns for name cheks) can be configured via

- --config command-line switches.
- <confin name=”n” value=”v”> entries in ant task definitions.
- preferences file. Example of such file can be found in **etc** subdirectory of JavaChecker distribution, or cited here:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE preferences SYSTEM "http://java.sun.com/dtd/preferences.dtd">
<preferences>
<root type="user">
 <map />
 <node name="ua" >
  <map />
    <node name="gradsoft" >
    <map />
      <node name="javachecker" >
       <map>
         <!-- JavaChecker config file for properties -->
         <!-- usage: JavaChecker -prefs <this-file>  -->
         <entry key="CheckEmptyCatchClauses" value="true" />
         <entry key="CheckGenericExceptionCatchClauses" value="true" />
         <entry key="CheckGenericExceptionSpecifications" value="true" />
         <entry key="CheckOverloadedEquals" value="true" />
         <entry key="CheckNamePatterns" value="true" />
         <entry key="NonFinalFieldNamePattern" value="[a-z].*_" />
         <entry key="FinalFieldNamePattern"
                      value="[A-Z]([A-Z]|[0-9]|_)*|serialVersionUID" />
         <entry key="ClassNamePattern" value="[A-Z].*" />
         <entry key="LocalVariableNamePattern"
                            value="[a-z]([A-Z]|[0-9]|[a-z]|_)*([A-Z]|[a-z]|
[0-9])" />
         <entry key="MethodNamePattern" value="[a-z].*" />
         <entry key="EnumConstantNamePattern"  value="([A-Z]|[0-9])+" />
         <entry key="CheckNonFinalPublicFields" value="true" />
       </map>
    </node>
   </node>
 </node>
</root>
</preferences>
```

As we see, names and values stored as properties of ua.gradsoft.javacheker package.

Attributes of source objects.

Source Annotations

Sometimes we needed some additional information about source objects, which can't be derived from source code. For example, look at process of checking unclosed closeable object: exists object, which inheried from Closeable, but really are safe to use without close ('StringWriter'). For this purpose JavaChecker provide set of source annotations. First is *TypeCheckerProperties*

```
@Retention(RetentionPolicy.SOURCE)
@Target(ElementType.TYPE)
public @interface TypeCheckerProperties {
      public String[] value()  default {};
}
```

values are just array of strings, which must contains odd number of entries, interpreted as key/value pairs.

Key can be any string, value parsed as term. Later this properties are accessible from context of java source processors via JavaTypeModel API (see setAttribute/getAttribute in JavaTypeModel interface)

Example of usage:

```
@TypeCheckerProperties({"NotCloseable","true","ThreadSafe","true"}
)
public class MyThreadSafeWriter extends Writer
{
 .....
}
```

will define two attributes of class MyThreadSafeWriter, both with value “true”.

Of course, not only types can be marked by attributes, so for this purpose exists set of simular annotations for constructors (*ConstructorCheckerProperties*), methods (*MethodCheckerProperties*) and fields (*FieldCheckerProperties*)

Such annotations are defined in jar JavaChecker2Annotations.jar (avaible in *lib* directory of distribution), sources are situated in *jsrc-ann* directory.

Source Annotations in external files.

Sometimes we need store additional information about third-party software package, where sources are not available or maintained by some external organization. For such cases exists approach to store model properties outside source code in files with extensions j*avacfg*.

The layout of *javacfg* file must be the same, as layout of appropriate source file (relative to *attrdir* configuration properties). Inside javacfg files must be java declarations of appropriate types, methods and fields. (Note, that information about inheritance and signatures of methods and fields, for which we does not set properties, can be omitted).

For example, let we want mark class java.io.StringWriter as noncloseable.

Them inside *attrs* directory we create subdirs *java/io* with file *StringWriter.javacfg*. Content of file is next:

```
package java.io;

@TypeCheckerProperties( { "NotCloseable","true" } )
class StringWriter {}
```

Note, that it is possible to set several root directories for external source properties, in such case distinct properties for same class will be 'summarized'.

Disabling checks from source code.

Also JavaChecker provide special annotations to disable specified types of checks via annotations.

Type annotation CheckerDisable is used for this purpose. Here is definition:

```
@Retention(RetentionPolicy.SOURCE)
@Target(ElementType.TYPE)
public @interface CheckerDisable {
       public String[] value() default {};
}
```

Vaue is string, where must be names of disabled annotations, separated by comma or “All”. I.e. @CheckerDisable({“GenericExceptionCatchClause”,”UnclosedCloseable” }) will disable checks whith those names for annotated type. @CheckerDisable({ “All” }) will disable all checks.

Old-fashion checker comments (as in 1.x versions) also works (but limited). Note, that all checks is disabled for automatically generated source files from JavaCC and byacc.

Some predefined checks

Naming

JavaChecker can perform style checking of identifiers, which based on role of identifier and regular expression pattern for this roles. This check is enabled by property *CheckNamePatterns* and by default follows the Sun Java Convention. Configuration properties are next:

- *ClassNamePattern* -- set pattern for names of classes, interfaces and enums. (default is [A-Z_]+.*)
- *MethodNamePattern* – set pattern for method names. (default is *[a-z]+([A-Z]|[0-9]|[a-z]|_)**
  - For names of variables and function parameters:
    - *NonFinalFieldNamePattern* – non-final fields. Default is `[a-z].*`
    - *FinalFieldNamePattern* – final fields. Default is `[A-Z](A-Z|_|0-9)*|serialVersionUID`
    - *LocalVaribleNamePattern* – local variables and formal parameters. Default is [a-z]+([A-Z]|[a-z]|_[0-9])*
    - *EnumConstantNamePattern* – for enumeration constants. Default is `[A-Z](A-Z|_|0-9)*`
  - *TypeArgumentNamePattern* – for name of type parameters. Default is [A-Z]+([A-Z]|[0-9])*

Misc. style checks.

Also exists some predefined style checks:

- ***CheckEmptyFile***

  if enabled, warn about empty java files in project.

- ***EmptyPackageName***

  warn about empty package name.

- ***NonFinalPublicFields***

  warn about non-final public fields.

Exceptions

- ***GenericExceptionCatchClause***

  if enabled, warn about catching of generic exceptions.

- ***EmptyCatchBlock***

  warn about empty catch block.

- ***GenericExceptionThrow***

  warn about throwing of generic exception.

Program flow

- ***SwitchLabelWithoutBreak***

  if enabled, warn about non-empty switch label without statement, which break control flow (i.e. break or throw or return). For example next chunk of code:

```
switch(x) {
  case 1:
  case 2:
       System.out.println("x<3");
        break;
    case 3:
    System.out.println("x<4");
    throw new Exception("qqq");
 case 4:
   System.out.println("x<5");
  case 5:
    System.out.println("x<6");
    return;
 default:
  System.out.println("x>=6");
```

```
            }
```

will trigger this violations for case block with label 4.

- ***SwitchWithoutDefault***

  if enabled, warn about switch statement without default label.

Resources usage

- ***UnclosedCloseable***

  warn about possible unclosed closeable.   I. e. For example if we open file in function and forgott about one.

  Example:

```
public static void main(String[] args)
{
 try {
       FileReader reader = new FileReader(args[1]);
       int ch=reader.read();
}catch(FileNotFoundException ex){
       ex.printStackTrace()
}catch(IOException ex){
       ex.printStackTrace();
}
}
```

This check depends from few properties, which can be set via anotations or as external:

- *NotCloseable* -- type property, which indicate, that instance of this class really can be safe forgotten without closing.
- *CloseableProxy* – type property, which indicate that instance of this class can be used as proxy to some other closeable. (Example – PrintWriter). In such case close call if checked when argument of allocation exception is another closeable allocation exception.
- *ResourceFactory* – method property, which indicate, that method create new resource, which must be closed.

- ***UnusedLocally***

  check warn about unused private method or field.

  (except special method and fileds of java.io.Serializable: serialVersionUID, readObject, writeObject, readObjectNoData)

Comparison and equality

- ***ObjectComparisonWithEquality***

When enabled, warn when objects are compared to other objects (other then null) using equality expression (ie via '==') instead equals.

- ***EqualsHashCode***

  Check, that if object has overloaded equals, than hashCode also must be overloaded.

More checks will be available with time. May be main value of javachecker is not in running basic set of checks, but in providing possibility for writing own domain-specifics tests in pluggable manners. For example, custom JavaChecker rules was using in project coin, to determinate: what language changes will have more effects to optimizing existing java codebase. (this rules are included in source distribution)

Advanced usage: write custom checks

General processing structure.

So, software check (or more general – any AST processing activity) can be represented as pluggable interface, which consists from name, type, description and rules. Rules are depends from type of processing (will be described later), name, type, description – just strings.

List of all external checks are situated in files with names *checkers_*.def* in */etc* subdirecotry of JavaChecker2 distribution.

Each check is described as check definition file, which looks as follows:

Example:

```
[
   define(CheckMyCompanyPackageName,
     #description
     "Check that names of packages are started from com.mycompany",
     #type
     BT_COMPILATION_UNIT_RULESET,
     #rules
     ruleset(
       PackageDeclaration(Name([$x,$y])->COM_MY_COMPANY([$x,$y],$x),
       COM_MY_COMPANY([Identifier(“com”),[Identifier(“mycompany”),$x]],$y) -> true
                                                   !-> false
       [violationDiscovered("Naming", "package does not start from com.mycompany",$y)]
      ),
      #enabled by default
      true
 );
]
```

Type must be one of:

- BT_COMPILATION_UNIT_RULESET -- rules in ruleset are applied to AST tree of compilation unit with bt-strategy. In short the work of rules is to determinate syntax patterns, reduce one and call special “violationDiscovered” action in case of finding problem (see TermWare documentation for details of different strategies). For structure of AST term refer to source code of Java5PP subproject (we receive tree structure as in JavaCC rules, than apply transformations, described in class ASTTransformers.java)

- BT_TYPE_RULESET -- rules in ruleset are applied to AST tree of each type definition in compilation unit with bottom-top strategy.
- FT_TYPE_RULESET -- rules in ruleset are applied to AST tree of each type definition in compilation unit with first-top strategy.
- JAVA_CLASS – model is passed to class, which implement appropriative callback interface. (*ua.gradsoft.checkers.JavaTypeModelProcessor*)
- MODEL_RULESET – rules are applied not to AST tree, but to term of semantics model of program. How model term differs from AST term – for each language element object model of appropriative semantics entity (class, expression, so. on) is created and model term contains not only syntax elements, but special reference to java object (named context) which give us access to semantic model of program: it is possible to resolve classes, clarifying local variables and so on. This allows build sophistical checkers which can do abstract interpritation of Java programs. For details, look at 'models' subpackage of JavaChecker.

And last entry is **true** or **false** which describe, that this check is enabled (or disabled) by default. We can enable/disable additional checks from command line or ant task or preferences.

So, in short, the process of adding new check for JavaChecker can be described by next steps:

- decide, what type of check you needed.
- write appropriative class or ruleset (using Java or TermWare as language and JavaChecker2 source code as main information source about representation of Java entities)
- add you check to etc/checkest.def (or provide own file).
- (optionally) -- publish somewhere :)

Excepts checks it is possible to define calculated statistical value characteristics of analyzed code. For each check we have statistical characteristic with same name: number of entries inside scope. Also we can define new calculated characteristics using 'calculate' term.

Example:

```
calculate(DiamondInitializers_percents,"coin",
        "per all variable initializers",
        (DiamondInitializers/AllVariableInitializers)*100,
        false),
```

This is part of set of checks for coin project, wich calculate value: how many variable initializers use allocation expressions with type arguments (and therefore can be optimized in case of accepting diamond initalizer proposal for project coin) among all other variable initializers.

So, first argument is name, second and third are category and descriptions (same as in checker definition), third is arithmetical expression, where other statistics values and counters can be referenced by name, last argument is used for enabling or disabling of calculating and printing of such value during processing.

For additional information you can look at:

- javachecker project wiki: http://redmine.gradsoft.ua/wiki/javachecker

- files buid-check-external and etc/checkers_*.def for examples of check defintions.
- API description: http://datacenter.gradsoft.ua/files/ua.gradsoft/javachecker/javadoc/

Document attributes.

Organization: Grad-Soft Ltd, Kiev, Ukraine. http://www.gradsoft.ua

Author: Ruslan Shevchenko Ruslan@Shevchenko.Kiev.UA

Version: 2.5.1

History of changes:

21.06.2009 version 2.5.1 (reviewed; added notes about using custom checks and statistics; added explicitEnableOnly, scopes and statistics, project coin checks).

29.07.2007 version 2.2.3 (added fork, jvmarg and failOnError parameters of ant task).

11.06.2007 version 2.2.2 (maintance release)

06.06.2007 version 2.2.1 (added CheckerDisabled annotation)

27.05.2007 version 2.2.0

(added work with properties, SwitchWithoutDefault and UnusedLocally tests. Rework of UnclosedCloseable)

21.04.2007 version 2.1.1 (added SwitchLabelWithoutBreak test)

15.04.2007 version 2.1.0

19.03.2007 version 2.0

07.06.2004 version 1.x

17.03.2004 initial revision.