Contemplate
THINKING ABOUT SOFTWARE

ThreadSafe Command Line Interface
User Guide

July 24, 2014

Contents

Introduction

The ThreadSafe Command Line Interface enables Java code to be analyzed via the command line. Results for each analysis are shown in generated HTML reports. ThreadSafe CLI provides:

- A set of concurrency related analyses.
- Generated HTML reports showing analysis *findings* in the source code.
- Full rule documentation via the HTML report.

System Requirements

ThreadSafe supports the following platforms:

- Windows XP or later
- Mac OS X 10.6 or later

- Linux (glibc 2.3 or later)

The software requirements include:

- Java SE 6 Update 10 or later

ThreadSafe does not require any special privileges to run. It can be installed in a normal user account or on a shared file system.

Distribution Contents

The software is distributed in a `zip` file. This should be extracted to a local directory on your machine. The distribution includes the following contents organized in an installation directory named `ThreadSafe-CLI-<version>`:

`Documentation` This directory contains this document, an example configuration file and the ThreadSafe license terms.

`lib` This directory contains third-party libraries and other dependencies used by this software.

`Licenses` This directory contains license agreements for third party libraries that are used by and included with this software.

`threadsafe.jar` This file is the command line interface `jar` archive which is used to run analyses and produce HTML reports.

`threadsafe.properties` This is the global configuration file of the command line interface.

Support

If you have any questions or comments, please contact us at support@contempleteltd.com.

Installing the ThreadSafe Command Line Interface

To install ThreadSafe, extract the contents of the distribution to the filesystem. The installation directory can be renamed if desired.

Ensure that the path to the Java executable has been added to the operating system's `PATH` environment variable. Instructions on how to do this should be available with the operating system.

To verify that the Java executable has been added, run the following command to print the Java version and check that it satisfies the system requirements:

```
java -version
```

The installation can be tested by printing the ThreadSafe help message with the following command:

```
java -jar <installation-directory>/threadsafe.jar -h
```

License Key

After installation, a license key must be configured before any analyses can be run. If you do not have a valid license key, please contact `sales@contemplateltd.com`.

To configure the license key:

1. Open the `threadsafe.properties` file in the installation directory with a text editor.
2. Set the value of the `licenseKey` property using copy and paste.
3. Save the changes.

Concepts

Rules

A *rule* is a description of a specific pattern of code that may lead to software faults. ThreadSafe checks code against the rules enabled in the rule configuration and creates findings for each violation found.

All rules have an associated name, category (e.g. *Locking* and *Collections*) and severity. Some rules have additional parameters. Rules can be enabled, disabled and have their parameters changed using rule configuration files (see Rule Configuration).

Full rule documentation is available in the HTML report. It can be accessed while investigating a finding by clicking the ⓘ Rule description link from the Detail View.

Findings

A *finding* is a description of a rule violation. Findings appear in the Findings View once analysis completes.

There may be multiple locations in the source code associated with a finding. Each finding has a *primary location* which identifies the main location of the violation. This may be a particular line of code such as a field declaration in a class. The HTML report provides links to primary locations. These are the first (uppermost) entries from the link list in the Detail View. In addition, the Source Code View initially sets focus on primary locations, highlighting them with a dark blue color.

Findings may also have *additional locations*. These provide more information which helps explain the violation. For example, additional locations may show where a field is read or written to, or where a lock is taken. The HTML report provides links to additional locations. These are all the links from the link list in the Detail View apart from the first (uppermost) one. The Source Code View initially highlights additional locations with a light blue color.

Severities

ThreadSafe defines the following severity levels for findings:

- **Blocker** - Blocking issues that must not be allowed into production code.
- **Critical** - Critical issues that should be addressed urgently.
- **Major** - Major issues that should be addressed eventually.
- **Minor** - Minor issues that are not a significant risk.
- **Info** - Suggestions and related information.

Tasks

Compile Java Sources with Debugging Information

Java code to be analyzed should be saved and compiled before running ThreadSafe. ThreadSafe requires sources to be compiled with debugging information in the bytecode, including file names and line numbers.

A simple way to check that the compiler is configured appropriately is to check that a stack trace thrown by the application includes the file names and line numbers for classes in the project. If file names and line numbers are not present, the build tool should be configured as described below.

Apache Ant If the project is built with Ant, users can include debugging information by adding a `debug="true"` attribute to the `<javac>` task element. There is no need to specify the `debuglevel` attribute but, if it is used, the value should include the `source` and `lines` options.

Apache Maven If the project is built with Maven, then debugging information will be included by default and no specific configuration is required.

Javac If compiling with `javac` directly, the `-g` parameter should be specified on the command line.

Further information should be available in the documentation of the build tool.

Running Analyses

The recommended way to run the ThreadSafe analysis is to create an analysis configuration file (see Configuring Analyses) named `threadsafe-project.properties` in the root directory of the project and then run ThreadSafe from this directory with no arguments.

ThreadSafe uses the Java properties file format for its configuration files. A full description is available in the java.util.Properties Javadoc API documentation.

An example configuration file is shown below:

```
projectName=myProject
sources=src/main/java
binaries=target/classes
outputDirectory=threadsafe-html
```

After the configuration file has been saved, the project can be analyzed by executing the commands:

```
cd <project-root-directory>
java -jar <installation-directory>/threadsafe.jar
```

The path to the configuration file can be specified using the `-c` option. This may be useful if the configuration file is not in the project directory. Additionally, all properties in the configuration files can be overridden with command line arguments, using the `-D` option.

For example, the command below allows running analysis when the configuration files are stored in a centralized directory:

```
java -jar <installation-directory>/threadsafe.jar \
    -c <analysis-configs-directory>/project-config.properties \
    -DbaseDirectory=<full-path-to-project-root-directory>
```

When ThreadSafe configuration is stored outside the project directory, it may be preferable to to specify the `baseDirectory` property inside the configuration file (in this case `project-config.properties`).

Investigating Results

ThreadSafe generates an HTML report after running the analysis. This report can be viewed by opening the `index.html` file in the output directory with a web browser.

Findings appear as a tree in the Findings View. They can be organized in groups using the **Group by** drop-down. Additional information regarding a finding can be seen by selecting it in the list. Details appear in the Detail View and relevant code is highlighted in the Source Code View.

Accesses View The Accesses View allows detailed investigation of findings that involve locking. It is designed to make spotting problems with locking strategies easy. You can quickly open this view for a particular finding from the Detail View by clicking on the Accesses link.

The view presents a table. It lists source code locations that are accesses to the field referenced by the finding. The table shows which guards are held for each field access. Clicking a link in the leftmost column focuses the Source Code View on the location in the code.

You can find more about how to use this view from the Accesses View section.

Finding Locations Each finding refers to a primary and possibly to one or more additional locations in the source code. The Detail View of the ThreadSafe View provides links to these locations. Clicking a link opens relevant code in the Source Code View and highlights the position with a dark blue color.

The Source Code View displays finding locations in a source file by highlighting the corresponding lines. Primary locations and the line currently on focus are highlighted with a dark blue color and additional locations are highlighted with a light blue color.

Configuring Analyses

The ThreadSafe Command Line Interface is configured using properties defined in properties files or passed on the command line. There are three precedence levels.

Properties defined in the global configuration file have the lowest precedence. Project configuration files have a higher precedence. Configuration options specified as command line arguments have the highest precedence. Properties with higher precedence override properties with lower precedence.

The sections below describe the properties commonly defined at each level of precedence.

Global Configuration Properties in the global configuration file apply to all analyses, unless otherwise overridden. This file must be located in the installation directory under the name `threadsafe.properties`. The table below describes the properties commonly defined in it.

Property	Type	Description
Mandatory:		
`licenseKey`	String	The license key.

Project Configuration The project configuration file is used to define properties that apply to a single project only, such as the project classpath.

This file is normally called `threadsafe-project.properties`, and saved in the project root directory. Alternatively, the `-c` command line parameter can be used to specify a different path.

The table below describes the properties commonly defined in this file.

Property	Type	Description
Mandatory:		
`binaries`	Paths	Directories/JARs containing bytecode to be analyzed.
`sources`	Paths	Directories containing sources of code to be analyzed.
`outputDirectory`	Path	Output directory for HTML report.
Optional:		
`baseDirectory`	Path	Project root directory.
`libraries`	Paths	Directories/JARs required on the project classpath.
`projectName`	String	Project name to show in the report.
`rulesFile`	Path	Rule configuration file.

Paths may be absolute or relative, and should be comma-separated when more than one is specified in a configuration value.

Note: Windows users should use `'/'` as a path delimiter because the backslash (`'\'`) is the escape character.

Relative paths in the configuration are considered relative to:

1. `baseDirectory` if set, or
2. the directory of the project configuration file if used, or
3. the current working directory.

If `baseDirectory` is a relative path, is considered relative to (2) or (3) above, in the specified order.

Multi-module project configurations can be created by appending all binary, source and relevant library paths for each module to the `binaries`, `sources` and `libraries` configuration properties. ThreadSafe will then analyze the modules together and create a combined report. Support for specifying individual module configurations separately is planned for a future release.

Command Line Properties Additional properties can be passed on the command line using the `-Dkey=value` option. These have the highest precedence and can be used to override configuration from the global or analysis-specific configuration file.

Rule Configuration By default, ThreadSafe uses all rules with a default configuration.

Rules can be turned on and off and rule parameters can be changed by using rule configuration files. The ThreadSafe Eclipse plug-in can be used to generate such files by exporting the rule configuration in the ThreadSafe preferences page.

To run ThreadSafe with a configuration that differs from the default, the `rulesFile` configuration property can be used to specify the path of a rule configuration file.

Suppressing Findings

It may be useful to suppress a finding if code review deems that the code is correct, or that it requires no further attention. Findings that are suppressed are no longer shown in results.

Marking regions of Java source code for which findings should be ignored is done by adding comments. These are specialized comments and must be of the form described below. This feature may be familiar to users who have used suppression comments in Checkstyle.

Using comments to suppress findings

Findings can be suppressed by adding the following comments to the code:

- Add a comment like this at the start of the section where findings should be suppressed:

```
// ThreadSafe: OFF
```

- Add a comment like this at the end of the section where findings should be suppressed:

```
// ThreadSafe: ON
```

Example: The code below checks if a lock has been taken before attempting to acquire the lock. ThreadSafe warns about this code and suggests that the calls to `isLocked()` and `lock()` should be replaced with a single call to `tryLock()`.

```
if (!lock.isLocked()) {
    lock.lock();
    try {
        x = x + 1;
    } finally {
        lock.unlock();
    }
}
```

The finding can be suppressed by adding comments as follows:

```
// ThreadSafe: OFF
if (!lock.isLocked()) {
    lock.lock();
    try {
        x = x + 1;
    } finally {
        lock.unlock();
    }
}
// ThreadSafe: ON
```

Findings with multiple locations

The code shown in the following example is identified as containing *inconsistent synchronization*; the field `x` is accessed without any synchronization in the `get()` method. However, the other accesses to `x` occur in synchronized methods, suggesting that the lack of synchronization in `get()` may be a bug.

```java
package shallowlock;

/**
 * Sample class for the rule "Inconsistent synchronisation".
 */
public class InconsistentSynchronization {

    private int x;

    public synchronized void increment() {
        x++;
    }

    public synchronized void decrement() {
        x--;
    }

    public int get() {
        return x;
    }
}
```

Figure 1: A finding with multiple locations.

ThreadSafe associates this finding with locations in the code. The line where the field **x** is declared is considered a primary location. Lines in the code indicating accesses to that field are considered additional locations.

To suppress this particular finding, it is sufficient to add comments only to its primary location.

Adding suppression comments to surround the primary location suppresses the finding and hides it from the results. The screenshot below shows the result of adding suppression comments to the code shown above and running ThreadSafe again. The finding is no longer shown.

```java
package shallowlock;

/**
 * Sample class for the rule "Inconsistent synchronisation".
 */
public class InconsistentSynchronization {

    // ThreadSafe: OFF
    private int x;
    // ThreadSafe: OFF

    public synchronized void increment() {
        x++;
    }

    public synchronized void decrement() {
        x--;
    }

    public int get() {
        return x;
    }
}
```

Figure 2: A finding that has been suppressed using comment markers.

Using @GuardedBy annotations

`@GuardedBy` annotations are a way of documenting the locks that must be used when accessing fields. If field declarations are decorated with `@GuardedBy` annotations, ThreadSafe can check that the appropriate locks are always acquired before any access to the annotated field is performed. This provides a more predictable way to ensure correct synchronization than ThreadSafe's heuristic based inconsistent synchronization analysis.

Here, we give a short introduction on how to use the `@GuardedBy` annotation in your code, and document the exact syntax that ThreadSafe supports. For more information on how to use `@GuardedBy` annotations effectively, please consult the book *Java Concurrency in Practice* by Goetz *et al.*

Including `@GuardedBy` in your project

In order to use the `@GuardedBy` annotation in your code, you must first include the necessary jar as a dependency for your project. There are two standard definitions of the `@GuardedBy` annotation in common use. The two definitions are equivalent, but are defined in different packages. ThreadSafe supports both.

- The newer, recommended, definition is in the `javax.annotation.concurrent` package, as defined by JSR305. The Maven Central Repository provides an example of how to add the jar as a project dependency for some popular build systems. See the 'Dependency Information' section of the 'Artifact Details' page. The jar can also be downloaded from the same page.
- The older definition is in the package `net.jcip.annotations`, named after the book *Java Concurrency in Practice (JCIP)*. The Maven Central Repository provides an example of how to add the jar as a project dependency for some popular build systems. See the 'Dependency Information' section of the 'Artifact Details' page. The jar can also be downloaded from the same page.

A small example

The following code demonstrates the use of the `@GuardedBy` annotation to document the assumption that all accesses to the field `counter` are synchronized by locking on the containing object (referred to as `"this"`).

```
import javax.annotation.concurrent.GuardedBy;

public class Counter {

    @GuardedBy("this")
    private int counter;

    public synchronized void increment() {
        counter++;
    }

    public synchronized int getValue() {
        return counter;
    }
```

```
}
```

In this class, the `increment()` and `getValue()` methods are both `synchronized`, so the locking strategy specified by the `@GuardedBy` annotation has been respected. ThreadSafe will accordingly not report any violations of the `@GuardedBy` annotation in this code snippet.

If either of the methods had **not** been declared as `synchronized`, then ThreadSafe would report that the `counter` field's `@GuardedBy` annotation has been violated, and will list the locations where the field has been accessed with and without synchronization.

Syntax reference

The list below shows the valid forms for the `@GuardedBy` parameter that ThreadSafe understands:

- `this`: the annotated field is guarded by a lock held on the enclosing instance.
- `fieldName` and `"this.fieldName"`: the annotated field is guarded by a lock held on the object referenced by the field `fieldName` in the same instance. If `fieldName` is not a field of reference type (i.e., it is of primitive type: `int`, `long`, `float`, or `double`), then the annotation is invalid. If `fieldName` is of a type that is an implementation of the `java.util.concurrent.locks.Lock` interface, then it is expected that the annotated field is guarded by the use of `fieldName.lock()` instead of `synchronized` blocks.
- `C.class`: the annotated field is guarded by a lock held on the class `C`.
- `C.fieldName`: the annotated field is guarded by a lock held on the object referenced by the *static* field `fieldName`. If `fieldName` is not a field of reference type (i.e., it is of primitive type: `int`, `long`, `float`, or `double`), then the annotation is invalid. If `fieldName` is of a type that is an implementation of the `java.util.concurrent.locks.Lock` interface, then it is expected that the annotated field is guarded by the use of `fieldName.lock()` instead of `synchronized` blocks.

Reference

The Toolbar

The toolbar provides controls which allow:

- Link navigation between the different parts of the report.
  - **Summary** - shows **Summary** page, collapses findings tree.
  - **Findings** - shows Findings View.
  - **Packages** - shows Package View.
- Grouping of findings via the **Group by** drop-down. Grouping is done by:
  - Rule type
  - Rule category

- Rule severity
- Resource (i.e. `.java` source files)

Summary Page

The summary page serves as an overview of the analysis report. It appears when the analysis report is opened for the first time or when the **Summary** link is clicked.

The summary page includes the following information:

- Project named as defined in the analysis configuration.
- ThreadSafe version used for the analysis.
- Date and time of the analysis.
- Total number of findings.

Findings View

The ThreadSafe Findings View displays findings in a tree. It appears on the left hand side of the HTML report. Selecting a finding from the tree opens detailed information about that finding in the Detail View and highlights relevant source code in the Source Code View. Findings in the tree have icons indicating severity.

Figure 3: Findings View

Detail View

The Detail View shows information about the currently selected finding. This includes a short description of the violated rule as well as a list of location links. Clicking these will open the respective finding locations in the Source Code View.

The Accesses link shown below the location links opens the Accesses View in a panel below the Source Code View.

The Rule description link shown below the location links opens the rule documentation in a panel below the Source Code View.

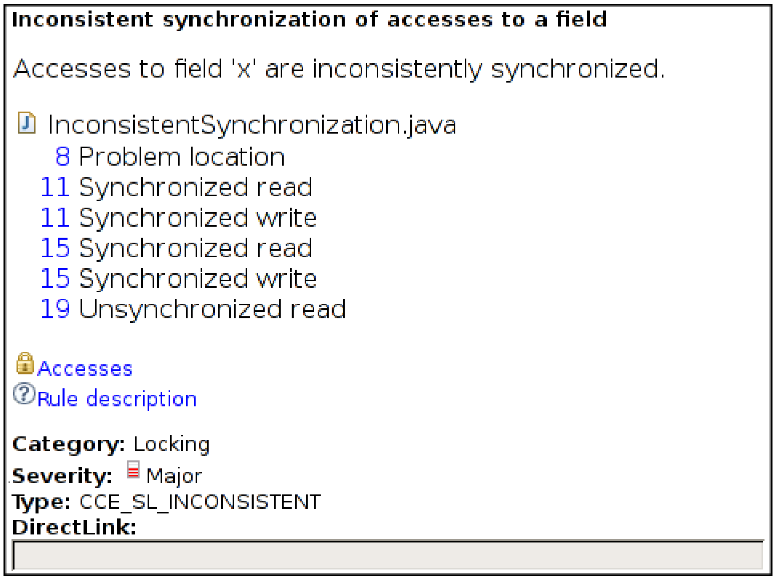

Figure 4: Detail View

Package View

The Package View allows browsing findings by navigating through source code packages and classes. The view is split in two - the Package View and the Class View. Clicking on a package in the Package View lists all its source files in the Class View. Clicking on a source file displays the source code in the Source Code View, highlighting all finding locations.

Source Code View

The Source Code View (see Figure 1) appears when a findings has been selected from the Findings View or a class has been selected from the Package View. It helps browsing findings by highlighting them in the source code.

The view features:

- Finding location highlighting
- Java syntax highlighting
- Cross-reference links
- Line numbering

Accesses View

The Accesses View shows all the potentially concurrent accesses to a shared field and how these are guarded by locks. It can be opened from the Detail View by clicking on the Accesses link.

The Accesses View is available for the following finding types:

- Inconsistent Synchronization
- Inconsistent Collection Synchronization
- Mixed Synchronization
- Mixed Collection Synchronization
- Thread-safe collection consistently guarded
- Unsynchronized write to field from asynchronous callback

The Accesses View will open showing accesses to the field referenced by the finding:

Guards for access to field InstanceGuards.shared:

	InstanceGuards.this.lock	InstanceGuards.this
InstanceGuards.java: 8	Not Held	Always Held
InstanceGuards.java: 13	Always Held	Not Held
InstanceGuards.java: 19	Always Held	Not Held

Figure 5: Accesses View

In this case, accesses to the field `InstanceGuards.shared` are shown. The field name is displayed at the top of the view.

A *guard* represents the relationship between a field access and a lock that is held during that access. This relationship is independent of how the accessed object and the lock are referenced. For example, in the following code, the accesses in the `getShared()` and `staticGetShared()` methods have the same relationship between the field and the lock: both access a field on a runtime object *obj* while holding a lock on the runtime object *obj*`.lock`. It doesn't matter that in the `getShared()` method, *obj* is referenced through the variable `this` and in the `staticGetShared()` method through the variable `instGuards`. Because the relationship between the accessed object and the lock is the same in each case, both accesses have the same guard: `InstanceGuards.this.lock`.

```
class InstanceGuards {

    private final Object lock = new Object();

    private Object shared;

    public synchronized void setShared(Object value) {
        this.shared = value; // Guarded by InstanceGuards.this
    }
```

```java

    public Object getShared() {
        synchronized (lock) {
            return shared; // Guarded by InstanceGuards.this.lock
        }
    }

    public static Object staticGetShared(InstanceGuards instGuards) {
        synchronized (instGuards.lock) {
            return instGuards.shared; // Guarded by InstanceGuards.this.lock
        }
    }
}
```

The Accesses View shows the accesses and guards for a finding's field, and is designed to make it easy to spot problems with locking strategies.

The view gives a table with a row for each access and a column for each guard. The first column provides access location information in the form of a link: source file and line number. A row shows which guards are *Always Held*, *Maybe Held* or *Not Held* for an access. If multiple accesses share the same *Always Held* guard, then ThreadSafe considers these accesses protected by a common lock.

A guard is *Always Held* for an access if every path leading to that access must take the guard lock. It is *Maybe Held* if the access occurs in a private or protected method, and that method is called both with and without the guard lock held. It is *Not Held* if there is no guarantee the guard lock will ever be held during the access.

Accesses View in Action

The following examples show how the Accesses View can help when diagnosing locking problems found by ThreadSafe.

Inconsistent Synchronization Inconsistent Synchronization findings report fields that are mostly, but not always, accessed while holding a common lock. The Accesses View shows which accesses are not guarded by a lock, and which lock has been used to guard the other accesses:

```java
class Inconsistent {

    private Object shared;

    public synchronized void set(Object value) {
        shared = value; // guarded access
    }

    public synchronized void unset() {
        shared = null; // guarded access
    }
```

```
    public Object get() {
        return shared; // unguarded access
    }

}
```

ThreadSafe will report an Inconsistent Synchronization finding on this code. The access to the `shared` field in `get()` is not synchronized while the rest are. The Accesses View shows the following:

Guards for access to field Inconsistent.shared:	
	Inconsistent.this
Inconsistent.java: 6	Always Held
Inconsistent.java: 10	Always Held
Inconsistent.java: 14	Not Held

Figure 6: Inconsistent guards

The guard `Inconsistent.this` is marked as *Always Held* in the `set()` and `unset()` methods and *Not Held* in the `get()` method . The unguarded access in `get()` immediately shows where the problem is. Clicking the access link in the leftmost column will focus the Source Code View on the location in the code.

Mixed Synchronization The Accesses View is especially useful for determining the cause of Mixed Synchronization findings, as shown from the InstanceGuards example in the screenshot below:

ThreadSafe will report a Mixed Synchronization finding on the InstanceGuards code above. The access to `shared` is guarded by two different locks: the InstanceGuards instance in the `set()` method and the `lock` object in the `getShared()` and `staticGetShared()` methods.

Guards for access to field InstanceGuards.shared:		
	InstanceGuards.this.lock	InstanceGuards.this
InstanceGuards.java: 8	Not Held	Always Held
InstanceGuards.java: 13	Always Held	Not Held
InstanceGuards.java: 19	Always Held	Not Held

Figure 7: Mixed Guards

From this we can see that accesses in the `getShared()` and `staticGetShared()` methods are guarded with `InstanceGuards.this.lock`, whereas the access in `setShared()` is guarded with `InstanceGuards.this`. To fix the problem, `setShared()` should synchronize on `InstanceGuards.this.lock` as shown below:

```
    public void setShared(Object value) {
        synchronized (lock) {
```

```
            this.shared = value;
        }
    }
```

Maybe Held Guard A common source of synchronization errors occurs when a method is sometimes invoked with a lock held and sometimes without. This is demonstrated by the following code:

```
class MaybeHeld {

    private Object shared;

    public synchronized Object get() {
        return shared; // Guarded by MaybeHeld.this
    }

    public synchronized void set(Object value) {
        privateSet(value); // Call privateSet() while holding MaybeHeld.this
    }

    public void setLater(long delay, final Object value) {
        Timer t = new Timer();
        t.schedule(new TimerTask() {
            @Override
            public synchronized void run() {
                privateSet(value); // Call privateSet() while holding MaybeHeld$1.this
            }
        }, delay);
    }

    private void privateSet(Object value) {
        shared = value; // Guarded by either MaybeHeld.this or MaybeHeld$1.this
    }

}
```

This code is similar to the mixed synchronization example above, except that the `set()` and `setLater()` methods call a private setter, `privateSet()` instead of writing to the `shared` field directly.

The `set()` method calls `privateSet()` with a guard held on `MaybeHeld.this`, but the `run()` method in the inner class calls `privateSet()` with a guard held on `MaybeHeld$1.this`. In this case, the Accesses View shows both guards as being *Maybe Held* for the access in `privateSet()`:

Guards for access to field MaybeHeld.shared:		
	MaybeHeld.this	MaybeHeld$1.this
MaybeHeld.java: 6	Always Held	Not Held
MaybeHeld.java: 24	Maybe Held	Maybe Held

Figure 8: Sometimes synchronized access

As no guard is *Always Held* in the method `privateSet()`, the access is considered unguarded and an error icon is shown in the second row.

Guard Types

ThreadSafe recognizes several different types of guards. These are described here.

Instance Guards A guard is an *instance guard* if the lock object is relative to the accessed object. Consider the following code again:

```
class InstanceGuards {

    private final Object lock = new Object();

    private Object shared;

    public synchronized void setShared(Object value) {
        this.shared = value; // Guarded by InstanceGuards.this
    }

    public Object getShared() {
        synchronized (lock) {
            return shared; // Guarded by InstanceGuards.this.lock
        }
    }

    public static Object staticGetShared(InstanceGuards instGuards) {
        synchronized (instGuards.lock) {
            return instGuards.shared; // Guarded by InstanceGuards.this.lock
        }
    }
}
```

The simplest instance guard is a lock on the accessed object, as in the `setShared()` method in the example above. The field `this.shared` is written with a lock held on `this`. This is shown in the Accesses View as guarded by `InstanceGuards.this`:

Guards for access to field InstanceGuards.shared:		
	InstanceGuards.this.lock	InstanceGuards.this
InstanceGuards.java: 8	Not Held	Always Held
InstanceGuards.java: 13	Always Held	Not Held
InstanceGuards.java: 19	Always Held	Not Held

Figure 9: Instance Guards

An Instance Guard can also refer to a lock field, as with the `getShared()` and `staticGetShared()` methods. In the `getShared()` method, `this.shared` is read with a lock held on `this.lock`. In the `staticGetShared()` method, `instGuards.shared` is read with a lock held on `instGuards.shared.lock`. Both these accessed object/lock pairs have the same relationship, so the Accesses View shows them both as guarded by `InstanceGuards.this.lock`.

Static Guards A guard is a *static guard* if the lock object is either a static field or a class instance. Consider the following code:

```java
class StaticGuards {

    private static final Object lock = new Object();

    private Object shared;

    public static synchronized Object getShared(StaticGuards sg) {
        return sg.shared; // Guarded by StaticGuards.class
    }

    public static void setShared(StaticGuards sg, Object value) {
        synchronized (lock) {
            sg.shared = value; // Guarded by StaticGuards.lock
        }
    }

}
```

The access in the `setShared()` method is locked using the static field `lock`. This is labelled in the Accesses View as `StaticGuards.lock`:

Guards for access to field StaticGuards.shared:		
	StaticGuards.class	StaticGuards.lock
StaticGuards.java: 8	Always Held	Not Held
StaticGuards.java: 13	Not Held	Always Held

Figure 10: Static Guards

Note that this differs from a lock held on an instance field, which would be labelled `StaticGuards.this.lock`.

The `getShared()` method is marked as both static and `synchronized`. Thus a lock is acquired on the class `StaticGuards` on entry to this method. ThreadSafe infers that the access to the field `shared` in the `getShared()` method is guarded by the lock held on the class, and so reports the guard as `StaticGuards.class`:

The same effect would be achieved if we had explicitly synchronized on the class literal `StaticGuards.class`, like so:

```
public static Object getShared(StaticGuards sg) {
    synchronized (StaticGuards.class) {
        return sg.shared;
    }
}
```

Unknown Guards Sometimes ThreadSafe is not able to determine the relationship between the object that is being accessed and the locks that are held during that access. In such cases, ThreadSafe will report that an *unknown guard* is held. These guards are labelled as `<unknown>` in the Accesses View, as can be seen in the screenshot below:

Guards for access to field UnknownGuards.shared:

	<unknown>	UnknownGuards.this.lock	UnknownGuards.this
UnknownGuards.java: 8	Always Held	Not Held	Not Held
UnknownGuards.java: 8	Not Held	Not Held	Always Held
UnknownGuards.java: 13	Not Held	Always Held	Not Held
UnknownGuards.java: 13	Always Held	Not Held	Not Held

Figure 11: Unknown guards

This Accesses View was generated by analysing the following example code with ThreadSafe. A mixed synchronization finding on the field `shared` is reported. This code demonstrates two cases where the relationship between the target object of an access and the lock held is unclear.

```
class UnknownGuards {

    private final Object lock = new Object();

    private Object shared;

    public synchronized void copyFrom(UnknownGuards ug) {
        this.shared = ug.shared;
    }

    public void copyTo(UnknownGuards ug) {
        synchronized (lock) {
```

```
                ug.shared = this.shared;
            }
        }

}
```

This class contains two methods; `copyFrom()` and `copyTo()`. In both methods the field `shared` has been potentially accessed on two objects: one referenced through `this`, and one referenced through the `ug` parameter. (ThreadSafe does not have enough information to determine for sure whether or not these two variables actually reference the same object).

In the method `copyFrom()`, a lock is acquired on the object referenced by `this`. For the access to `this.shared`, the guard is `UnknownGuards.this`, as indicated by the *Always Held* in the second row in the Accesses View screenshot above. For the field access `ug.shared`, there is no obvious relationship between `ug` and the lock on `this`. Therefore, ThreadSafe reports the inferred guard as `<unknown>`, as shown in the first row of the Accesses View.

The case demonstrated in the method `copyTo()` is similar, except that a lock is held on the object in the field `lock` rather than on `this`. Again, there is no obvious connection between the lock held on `this.lock` and the field `ug.shared`, so an unknown guard is reported.

ThreadSafe will also report `<unknown>` guards in cases where it has been unsuccessful in tracking references to objects. A common case where this happens is in code that traverses linked data structures. Fortunately, such cases are rare, as most code uses standard library data structures like `java.util.LinkedList` rather than custom implementations.