JDT Enhancement: *Promote All Warnings to Errors*

Objective

Give the user the ability to turn all Java code warnings of a project into errors, specifically in a way that

- is simple and quick ("flip a single switch")
- preserves the ability to use @SuppressWarnings in the code to bypass specific warning (error) instances

Use Case

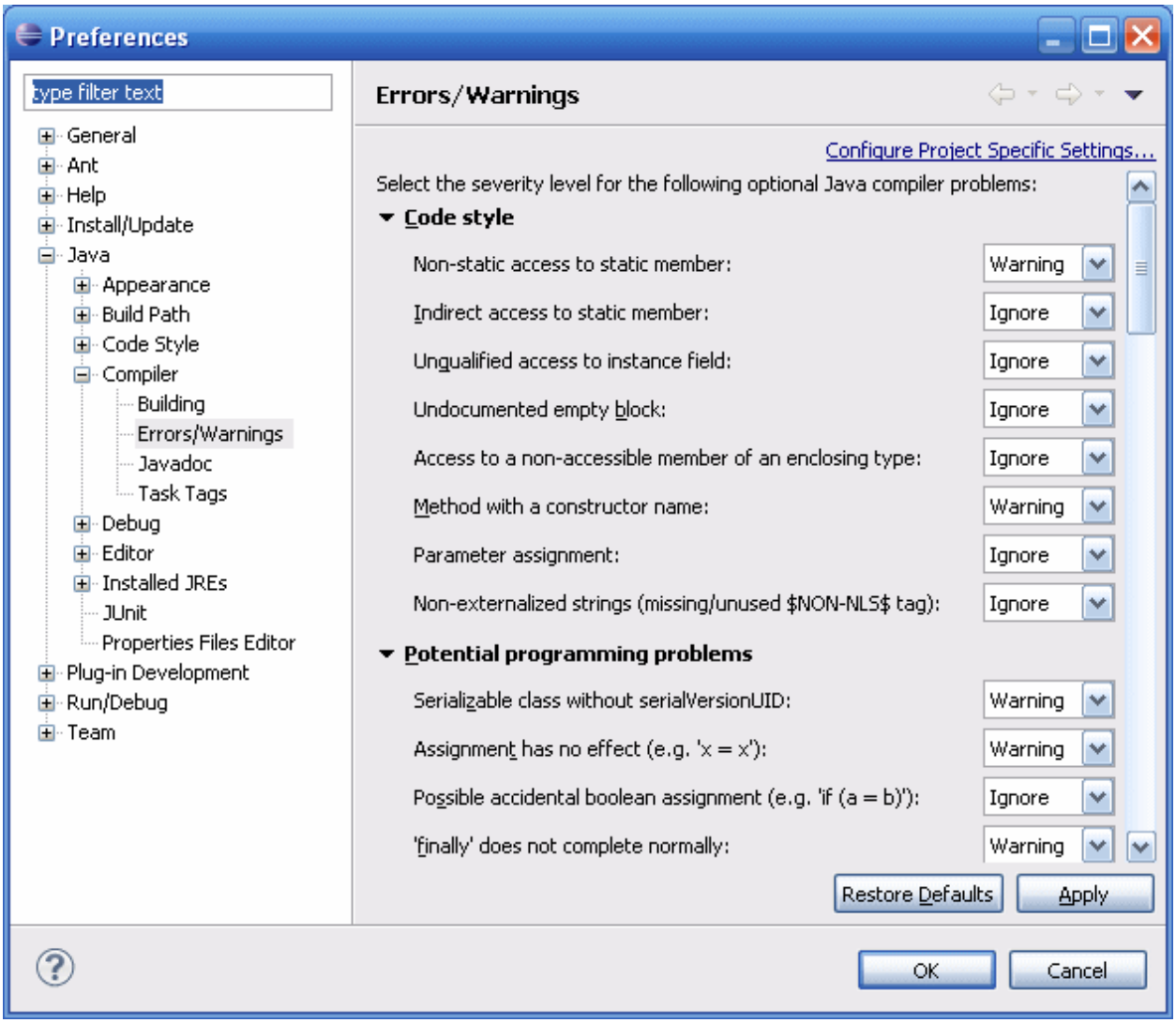
In a code quality improvement initiative, Joe Programmer has been tasked with ridding his team's codebase of warnings. His team has 40 Eclipse plugin projects in development, currently generating some 2000+ warnings. Developers have been complaining that the warning mechanism has been rendered useless by the magnitude of outstanding warnings, since they leave no practical way to determine if new code or changes in existing code are resulting in new warnings.

Joe spends three days cleaning up the code base and gets the warning count to zero.

Joe wants to make sure he never has to spend another three days fixing warnings. However, he doesn't want to continuously check the codebase, fixing new warnings himself or harassing the developers who introduce them. Joe wants the development tools to prevent their introduction. Developers can get away with checking-in code that has warnings, but not code that has errors, as that would break the build and draw the ire of co-workers and management. Joe talks to his manager and gets approval to configure his team's projects to treat all warnings as errors, as Joe knows that every error can be avoided one way or another--either by improving the problematic code or by using a @SuppressWarnings as a last resort. He goes through each project and enables a single checkbox that promotes all warnings to errors. The codebase is now self-checking and is set to remain warning-free indefinitely.

Deficiencies in JDT Today

JDT allows a wealth of coding patterns and situations to be treated as a warning or an error. There are some 50+ options available to the developer. These options are exposed in the *Java* > *Compiler* > *Errors/Warnings* preference/properties page. By default, a good number are set to *Ignore* and nearly all the others are set to *Warning*.



The first obstacle for Joe is that there's no easy way to change all the *Warning* option values to *Errors*. Not only are there 50+ options per project, but he has to carry out the changes for the 40 projects his team develops. Joe is resourceful, though, and realizes he can bypass the GUI and just tweak the *.settings/org.eclipse.jdt.core.prefs* file contents. With a simple global find-n-replace in his favorite text editor, he goes through and converts all occurrences of `"=warning"` to `"=error"`. It's more cumbersome

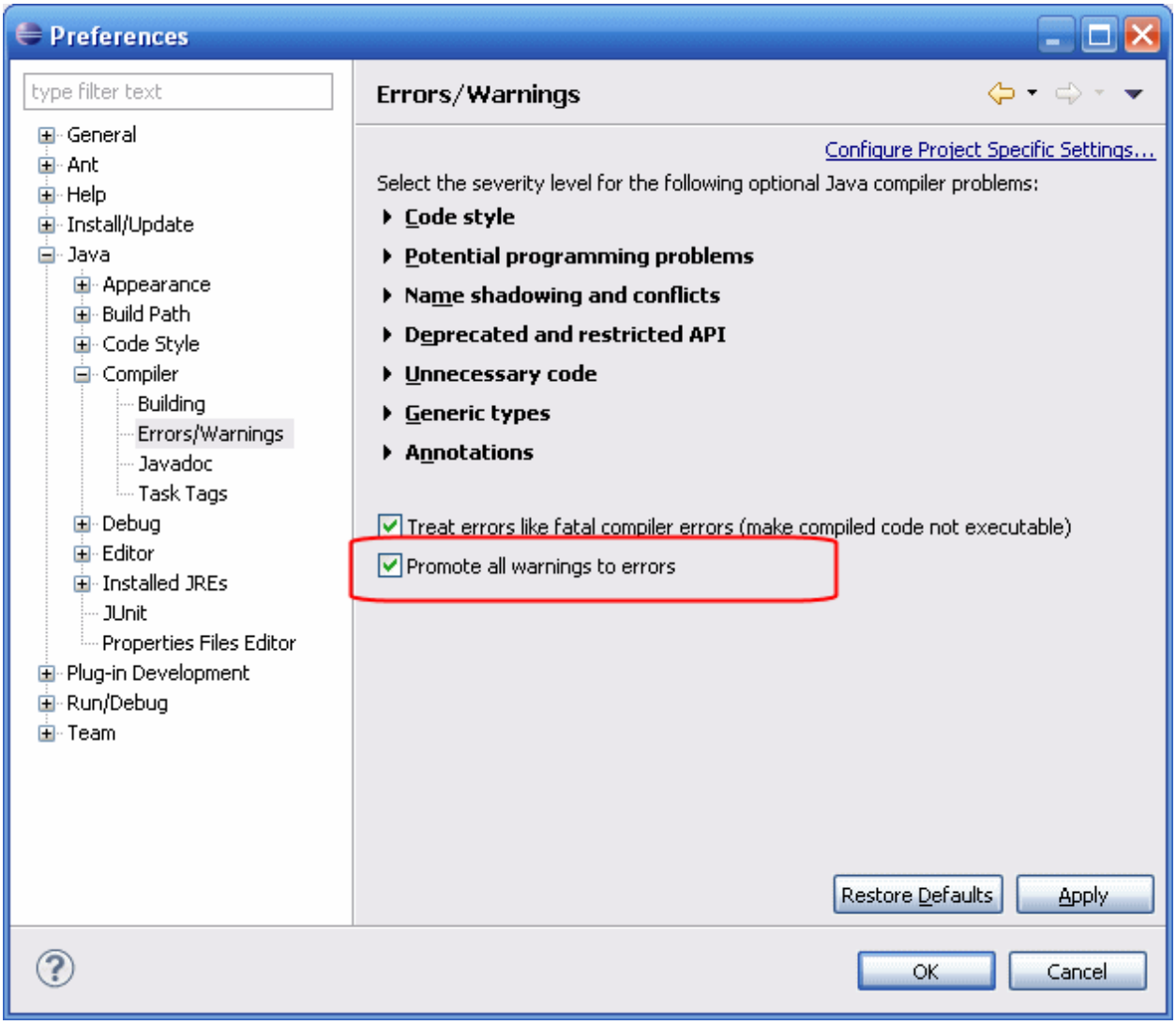
than flipping a single switch in the GUI, but it beats manually tweaking thousands of widgets.

But after adjusting a few projects, Joe notices something very troublesome. What used to be an empty Problems view is now a view filled with thousands of *errors*! It doesn't take Joe very long to realize what's wrong. Thousands of @SuppressWarnings in the code have just been rendered ineffective by having told JDT that many code patterns/ situations should be considered errors instead of warnings.

What Joe needs is a way to tell JDT to promote warnings into errors, but to still treat them as warnings when it comes to suppressing them with mark-ups in the code. And Joe would really like it if he could do that with a "single switch" solution, both globally and per project.

Solution Overview

I propose that we introduce a new option in the Java compiler options page



When the option is turned on, every Java code warning is automatically promoted up to an error. But these promoted warnings are marked such that they can be treated a little differently than a coding pattern/situation that has explicitly been set to generate an error. The only special treatment is that @SuppressWarnings will be honored in one case but not the other. Specifically, the annotation will be honored if the coding pattern/situation option is set to generate a warning, even though all warnings are being promoted to errors. However, coding patterns/situations which are explicitly marked to generate an error will NOT be silenced by the @SuppressWarnings. Also, the Quick Fix candidates should include a suppression fix when the error is one that has been promoted up from a warning, but not otherwise.

Warnings that are promoted to errors appear in the GUI as errors. There are nothing that visually differentiates them from "regular" (or "genuine") errors.

By default, the new option is turned off.

Like all other Java compiler options, the option has a global value which can be overriden at the project level.

Solution Details

A fully functioning solution is being submitted along with this proposal document in Bugzilla.

The new attribute associated with the preference is

```
org.eclipse.jdt.core.compiler.problem.promoteWarningsToErrors
```

and it has a value of `enabled` or `disabled`. If the attribute is not present, `disabled` is the default behavior.

At the core of the solution is an extension (new method) to IProblem to indicate whether the problem is an error that was promoted up from a warning (i.e., a "promoted warning"). A promoted warning is an error. A distinction between that sort of error and an error resulting from an explicit 'Error' option setting must be made in order to allow @SuppressWarnings to be effective on the former but not the latter.

IProblem is a public interface but it is marked as `noextend` and `noimplement`, thus we can add to it without a major version bump. IProblemLocation is equally extended. That extension is needed in order to propagate the "promoted warning" problem nature to the logic that assembles the quick-fix list, as it needs to include/exclude "Add @SuppressWarnings" based on whether the error is a promoted warning or not.

Unit Tests

Here we describe 15 tests which will help validate the feature

Prerequisite activity and assumptions:

- Create a new workspace
- Create a Java project using the New Project wizard.
- Create a class with a main() method.
- These tests are written with the assumption that they will be exercised in the sequence listed, and with no cleanup other than what's explicitly stated

TEST1
1. Examine global preference *Java* > *Compiler* > *Errors/Warnings* > *Promote all warnings to errors*. This pref will here-forth be referred to as "PAWTE".
Expected outcome: option should be unchecked. This test validates that the default state for the new preference is off.

TEST2
1. Set global preference *Java* > *Compiler* > *Errors/Warnings* > *Unnecessary code* > *Local variable is never read* to `Ignore`. This pref will here-forth be referred to as "LVINR".
2. Add a simple "int i;" to the main() method and save the file.
Expected outcome: Project rebuilds. **No issues** appear in the Problems view.

TEST3
1. Set LVINR to `Warning`
Expected outcome: Project rebuilds. A **warning** appears in the Problems view.
2. Right click on the problem in Problems view and select *Quick Fix*
Expected outcome: A list of available fixes appears and "Add @SuppressWarnings" **IS** one of them.
3. Apply the suppress warnings fix and save the java file.
Expected outcome: Item in Problems view goes away
Cleanup for next step: undo the fix (perform an undo in the editor)

TEST 4
1. Set LVINR to `Error`
Expected outcome: Project rebuilds. An **error** appears in the Problems view.
2. Right click on the problem in Problems view and select *Quick Fix*
Expected outcome: A list of available fixes appears and "Add @SuppressWarnings" **IS NOT** one of them.

TEST 5, 6, 7: repeat tests 2, 3, 4 but manipulating the LVINR project override instead of the global pref

TEST 8:
1. Turn off project overrides
2. Set LVINR in global prefs to Ignore
3. Enable PAWTE
Expected outcome: Project rebuilds. **No issues** appear in the Problems view.

TEST 9:
1. Set LVINR to `Warning`
Expected outcome: Project rebuilds. An **error** appears in the Problems view.
2. Right click on the problem in Problems view and select *Quick Fix*
Expected outcome: A list of available fixes appears and "Add @SuppressWarnings" **IS** one of them.
3. Apply the suppress warnings fix and save the java file.
Expected outcome: Item in Problems view goes away
Cleanup for next step: undo the fix (perform an undo in the editor)

TEST 10:
1. Set LVINR to `Error`
Expected outcome: Project rebuilds. An **error** appears in the Problems view.
2. Right click on the problem in Problems view and select *Quick Fix*
Expected outcome: A list of available fixes appears and "Add @SuppressWarnings" **IS NOT** one of them.

TEST 11, 12, 13: repeat 8, 9, 10 but manipulating the LVINR project override instead of the global pref

TEST 14, 15: repeat 12 and 13 but instead of selecting quick fix in the context menu of the Problems view element, right click on the problem marker in the Java editor and select quick fix there. Ensure that Add @SuppressWarnings is available or unavailable as per expectations in the steps 11 and 12