Code Inspection

IntelliJ IDEA features robust, fast, and flexible static code analysis. It detects compiler and runtime errors, suggests corrections and improvements before you even compile.

In this section:

Inspection Profiles

When you inspect your code, you can tell IntelliJ IDEA which types of problems you would like to search for and get reports about. Such configuration can be preserved as an *inspection profile*.

An inspection profile defines the types of problems to be sought for, i.e. which code inspections are enabled/disabled and severity of these inspections. Profiles are configurable in the Inspections settings page.

To set the current inspection profile (the one that is used for the on-the-fly code analysis in the editor), simply select it in the Inspections settings page and apply changes. When you perform code analysis or execute a single inspection, you can specify which profile to use for each run.

Inspection profiles can be applicable on the IDE or on the project level:

- **Project profiles** are shared and accessible for the team members via VCS. They are stored in project directory: `<project>/.idea/inspectionProfiles`.
- **IDE profiles** are intended for personal use only and are stored locally in XML files under the `USER_HOME/.<IntelliJ IDEA version>/config/inspection` directory.

> If necessary, you can change the applicability of the profile (current project of IDE) and, consequently, its location with the **Share profile** check box (selected means current project) in the Inspections settings page.

IntelliJ IDEA comes with the following pre-defined inspection profiles:

- *Default*: This local (IDE level) profile is intended for personal use, apply for all projects, and is stored locally in the file `Default.xml` under the `USER_HOME/.<IntelliJ IDEA version>/config/inspection` directory.
- *Project Default*: When a new project is created, the *Project Default* profile is copied from the settings of a template project. This profile is shared and apply for the current project. After the project is created, any modifications to the project default profile will pass unnoticed to any other project.
  When the settings of the *Project Default* profile are modified in the Template Project settings, the changed profile will apply to all newly created projects, but the existing projects will not be affected as they already have the copy of this profile.
  Project Default profile is stored in the `Project_Default.xml` file located in the `<project>/.idea/inspectionProfiles` directory.

One can have as many inspection profiles as required. There are two ways of creating new profiles: you can add a new profile as a copy of the Project Default profile or copy the currently selected profile. The newly created profiles are stored in XML files, located depending on the type of the base profile. .

The files `<profile_name>.xml` representing inspection profiles appear whenever some changes to the profiles are done and applied. The files only store differences against the default profile.

In case of the file-based project format, the shared profiles are stored in the project file `<project name>.ipr`.

Refer to the section Customizing Profiles for details.

Inspection Severity

Inspection severity indicates how seriously the code issues detected by the inspection impact the project and determines how the detected issues should be highlighted in the editor. By default, each inspection has one of the following severity levels:

- *Server problem*
- *Typo*
- *Info*
- *Weak Warning*
- *Warning*
- *Error*

You can increase or decrease the severity level of each inspection. That is, you can force IntelliJ IDEA to display some warnings as errors or weak warnings. In a similar way, what is initially considered a weak warning can be displayed as a warning or error, or just as info.

The color and font style used for highlighting of each severity level is configurable. Even more, you can create custom severity levels and set them for specific inspections.

If necessary, you can set different severity levels for the same inspection in different scopes.

> All modifications to inspections mentioned above are saved in the inspection profile currently selected in the inspection settings and apply when this profile is used.

For more information and procedural descriptions, see Configuring Inspection Severities.

Inspection Scope

By default all enabled code inspections apply in all project files. If necessary, you can configure each code inspection (enable/disable, change its severity level and options) individually for different scopes. Such configurations, like any other inspection settings, are saved and applied as part of a specific profile.

There may be complicated cases when an inspection has different configurations associated with different scopes. When such inspection is executed in a file belonging to some or all of these scopes, then the settings of the highest priority scope-specific configuration are applied. The priorities is defined by the relative position of the inspection's scope-specific configuration in the inspection settings: the uppermost configuration has the highest priority. The configuration **Everywhere else** always has the lowest priority.

For more information and procedural descriptions, see Configuring Inspection for Different Scopes.

Examples of Code Inspections

In the IntelliJ IDEA's Inspections settings page, you will find that all inspections are grouped by their goals and sense. Every inspection has an appropriate description. The most common tasks that are covered by the static code analysis are:

- Finding probable bugs.
- Locating dead code.

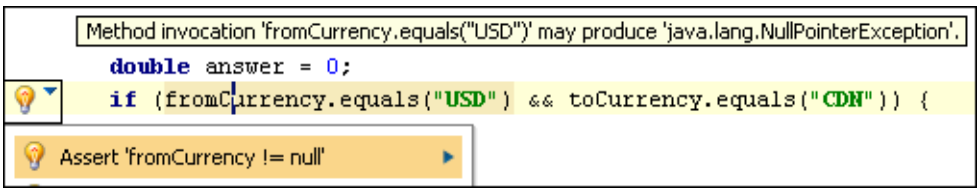
- Detecting performance issues.
- Improving code structure and maintainability.
- Conforming to coding guidelines and standards.
- Conforming to specifications.

Finding probable bugs

IntelliJ IDEA analyzes the code you are typing and is capable of finding probable bugs as non *compilation errors* right on-the-fly. Below are the examples of such situations.

Example. Potential NPE that can be thrown at runtime

Before



Here the first **if** condition may lead to a NullPointer exception being thrown in the second **if**, as not all situations are covered. At this point adding an assertion in order to avoid a NullPointer being thrown during the application runtime would be a good idea.

After

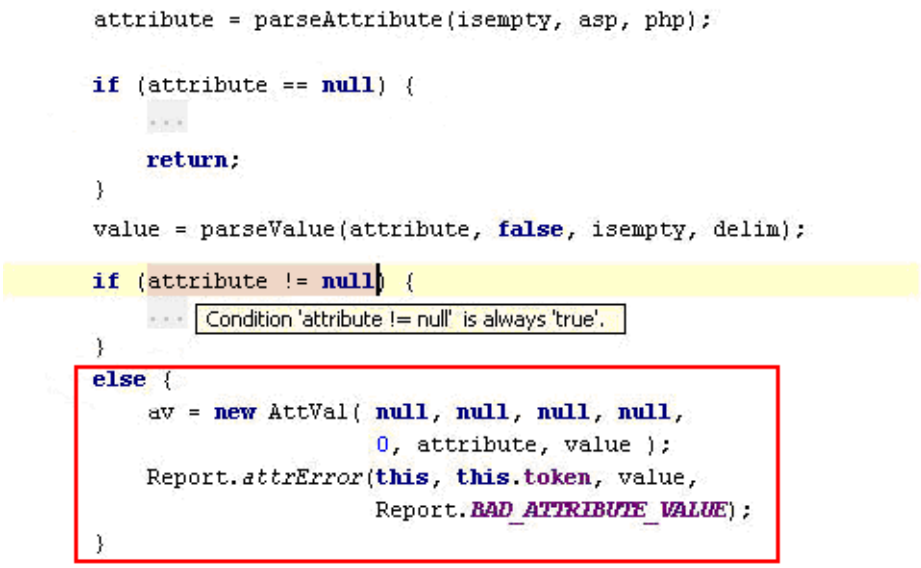
```
double answer = 0;
assert fromCurrency != null;
if (fromCurrency.equals("USD") && toCurrency.equals("CDN")) {
```

So, this is exactly what we get from the intention action.

Locating dead code

IntelliJ IDEA highlights in the editor pieces of so-called *dead* code. This is the code that is never executed during the application runtime. Perhaps, you don't even need this part of code in your project. Depending on situation, such code may be treated as a bug or as a redundancy. Anyway it decreases the application performance and complicates the maintenance process. Here is an example.

So-called *constant conditions* - conditions that are never met or are always *true*, for example. In this case the responsible code is not reachable and actually is a *dead* code.



IntelliJ IDEA highlights the if condition as it's always true. So the part of code surrounded with else is actually a dead code because it is never executed.

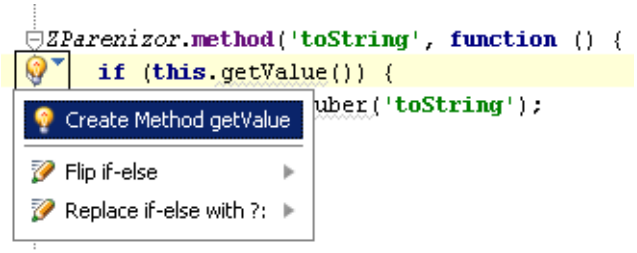
Highlighting unused declarations

IntelliJ IDEA is also able to instantly highlight Java classes, methods and fields which are unused across the entire project via **Unused declarations** inspection. All sorts of Java EE `@Inject` annotations, test code entry points and other implicit dependencies configured in the Unused declarations inspection are deeply respected.

For more examples of code inspections use, refer to
http://www.jetbrains.com/idea/documentation/static_code_analysis.html

Unresolved JavaScript Function or Method

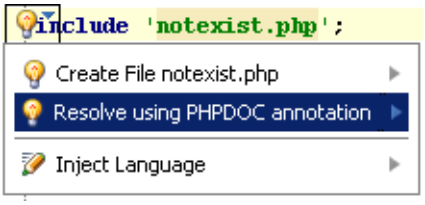
This inspection detects references to undefined JavaScript functions or methods.



Examples of PHP Code Inspections

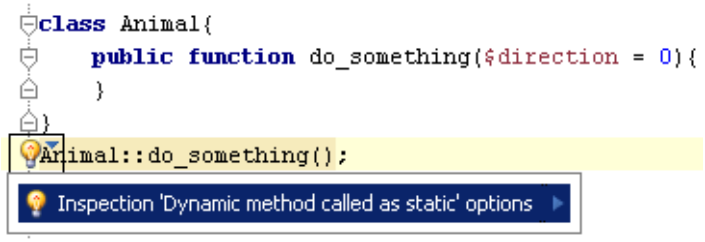
Unresolved include

This inspection detects attempts to include not actually existing files and suggests two quick fixes: to create a file with the specified name or use a PHPDOC annotation.



Dynamic method is called as static

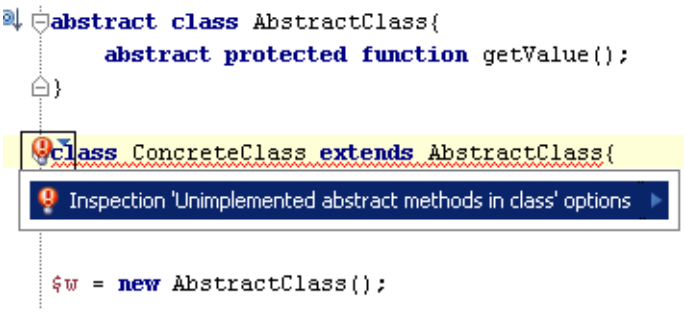
This inspection checks whether a call to a static function is actually applied to a static function.



The function `do_something()` is called as static while actually it is dynamic.

Unimplemented abstract method in class

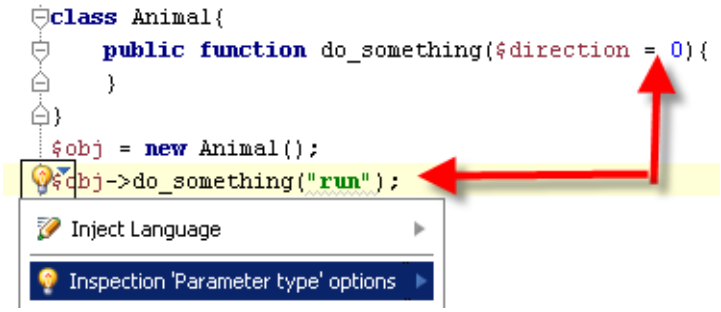
This inspection checks whether classes inherited from abstract superclasses are either explicitly declared as abstract or the functions inherited from the superclass are implemented.



The class `ConcreteClass` is inherited from an abstract class `AbstractClass` and has not been explicitly declared as abstract. At the same time, the function `GetValue()`, which is inherited from `AbstractClass`, has not been implemented.

Parameter type

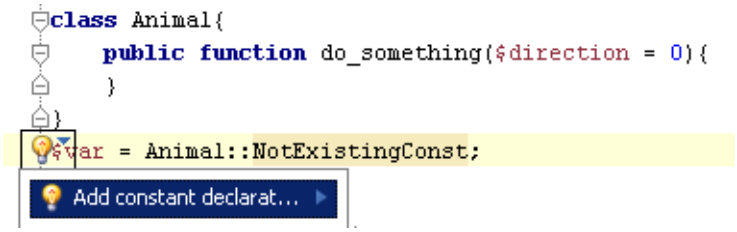
PHP variables do not have types, therefore basically parameter types are not specified in definitions of functions. However, if the type of a parameter is defined explicitly, the function should be called with parameters of the appropriate type.



The function `do_something` has the parameter of the type `integer` but is called with a `string`.

Undefined class constant

This inspection detects references to constants that are not actually defined in the specified class.



The constant `NotExistingConst` is referenced to as a constant of the class `Animal`, while actually it is not defined in this class.

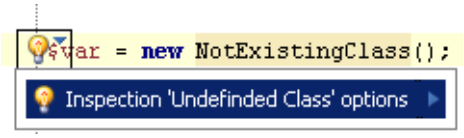
Undefined constant inspection

This inspection detects references to constants that are not actually defined anywhere within the inspection scope.

The referenced constant `UndefinedConst` is not defined anywhere within the inspection scope.

Undefined class

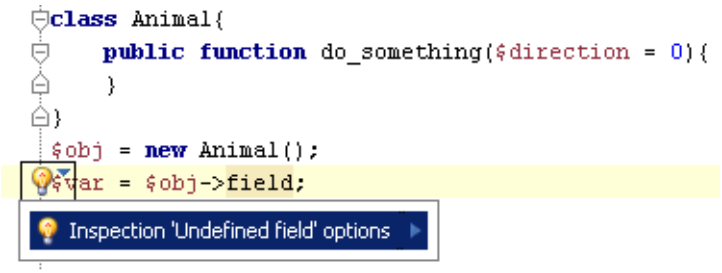
This inspection detects references to classes that are not actually defined anywhere within the inspection scope.



The referenced class `NotExistingClass` is not defined.

Undefined field

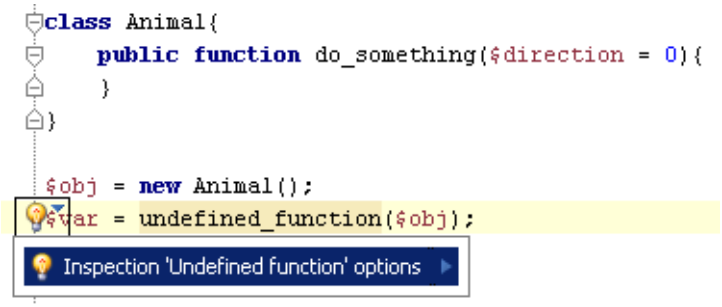
This inspection detects references to fields of a class that are not actually defined in this class.



The `$obj` variable is an instance of the class `Animal`. The declaration of the `$var` contains a reference to the `field` field of the class `Animal`, which is not defined on this class.

Call of undefined function

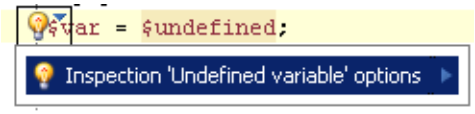
This inspection detects references to functions that are not defined anywhere within the inspection scope.



The called function `undefined_function()` is not defined anywhere within the inspection scope.

Undefined variable

This inspection detects references to variables that are not declared and initialized anywhere within the inspection scope. PHP does not require that each variable should be declared and initialized. PHP can initialize such variable on the fly and assign it the zero value. However, this inspection allows you to detect discrepancies of this kind.



In the *PHP* context, the *Undefined field* and *Undefined method* inspections may erroneously report severe problems when actually no problems take place. This happens when you attempt to access a property or to assign a value to a property that is not explicitly defined while the referenced class contains the `_get()` or `_set()` *magic* methods. No error should be reported because these methods are invoked every time an undefined property is referenced, however, IntelliJ IDEA still treats them as errors or warnings, depending on the severity you have specified for the inspection in general.

To suppress reporting errors in such cases, re-configure the inspection severity. To do that, open the Inspections page of the **Settings** dialog box, click the inspection name in the list and select the **Downgrade severity if __magic methods are present in class** check box in the **Options** area. After that undefined properties in such cases will be indicated one step lower than specified fir inspections in general, by default, *Info* instead of *Warning*.

See Also

Procedures:

- Intention Actions

Reference:

- Inspections
- Scopes
- Inspection Tool Window

External Links:

- http://www.jetbrains.com/idea/documentation/static_code_analysis.html
- Check lambda support in IntelliJ IDEA 12 EAP
- IntelliJ IDEA Inspections List

Web Resources:

- Developer Community