

# Refactoring Basics

[Previous](#) | [Quick Popups](#)

[Top](#) | [Quick Start](#)

[Next](#) | [Folding](#)



## Redirection Notice

This page will redirect to <https://www.jetbrains.com/idea/help/refactoring-source-code.html>.

IntelliJ IDEA offers a comprehensive set of automated code refactorings that lead to significant productivity gains when used right. This tutorial will teach you how to do that, starting from the basic topics.

## 1. Selection

First of all, don't bother selecting anything before you apply a refactoring. IntelliJ IDEA is smart enough to figure out what statement you're going to refactor, and only asks for confirmation if there are several possible choices.

```
import ...

public class UserTest {
    private User user;

    @Before
    public void setUp() throws Exception {
        user = new User();
    }

    @Test
    public void testToString() throws Exception {
        user.setEmail("demo@jetbrains.com");
        Assert.assertEquals("demo@jetbrains.com", user.toString());
    }
}
```

## 2. Undo

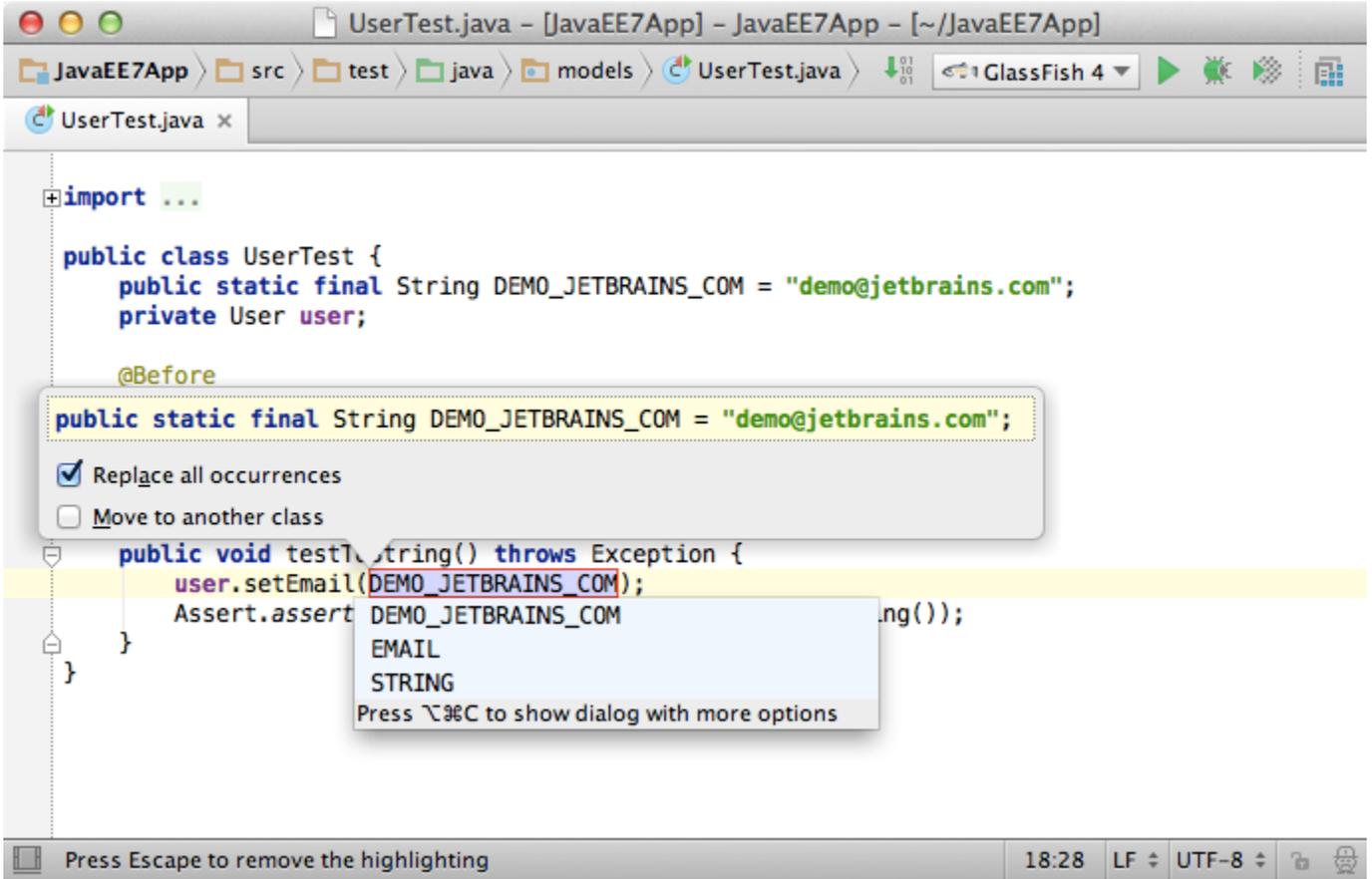
Another important thing to know is that IntelliJ IDEA lets you undo everything, and that includes refactorings, no matter how complex code transformations they cause. Just press `Ctrl+Z` (`Cmd+Z` for Mac), and you're back to where you were before inadvertently scrambling a few thousands lines of code.

## 3. Mnemonics

IntelliJ IDEA encourages you to use the keyboard instead of the mouse. It's proven to be faster and in the end will make you more productive at writing and transforming your code.

One thing that helps you easily use the keyboard to work with dialog boxes and popup windows is mnemonics---shortcut keys that are automatically assigned to each of the dialog elements. After you open a dialog, press and hold `Alt` to let IntelliJ IDEA

highlight all available mnemonics. Then you can use them by pressing the highlighted key while holding Alt to access the elements you need.



#### 4. String fragments

A real time saver is the ability to extract a part of a string expression with the help of the Extract...refactorings. Just select a string fragment and apply a refactoring to replace all of the selected fragment usages with the introduced constant or variable.

```
package models;

import ...

public class UserTest {
    private User user;

    @Before
    public void setUp() throws Exception {
        user = new User();
    }

    @Test
    public void testToString() throws Exception {
        user.setEmail("demo@jetbrains.com");
        Assert.assertEquals("demo@jetbrains.com", user.toString());
    }
}
```

Press Escape to remove the highlighting

1:14 LF UTF-8

## 5. Change variable type

Note that you can select a variable type when using the Extract variable refactoring. Press Shift+Tab when editing the variable name, and IntelliJ IDEA will offer you to select the variable type (e.g., you can tell it use the interface instead of implementation, or vice versa).

```
import ...

public class UserTest {
    private User user;

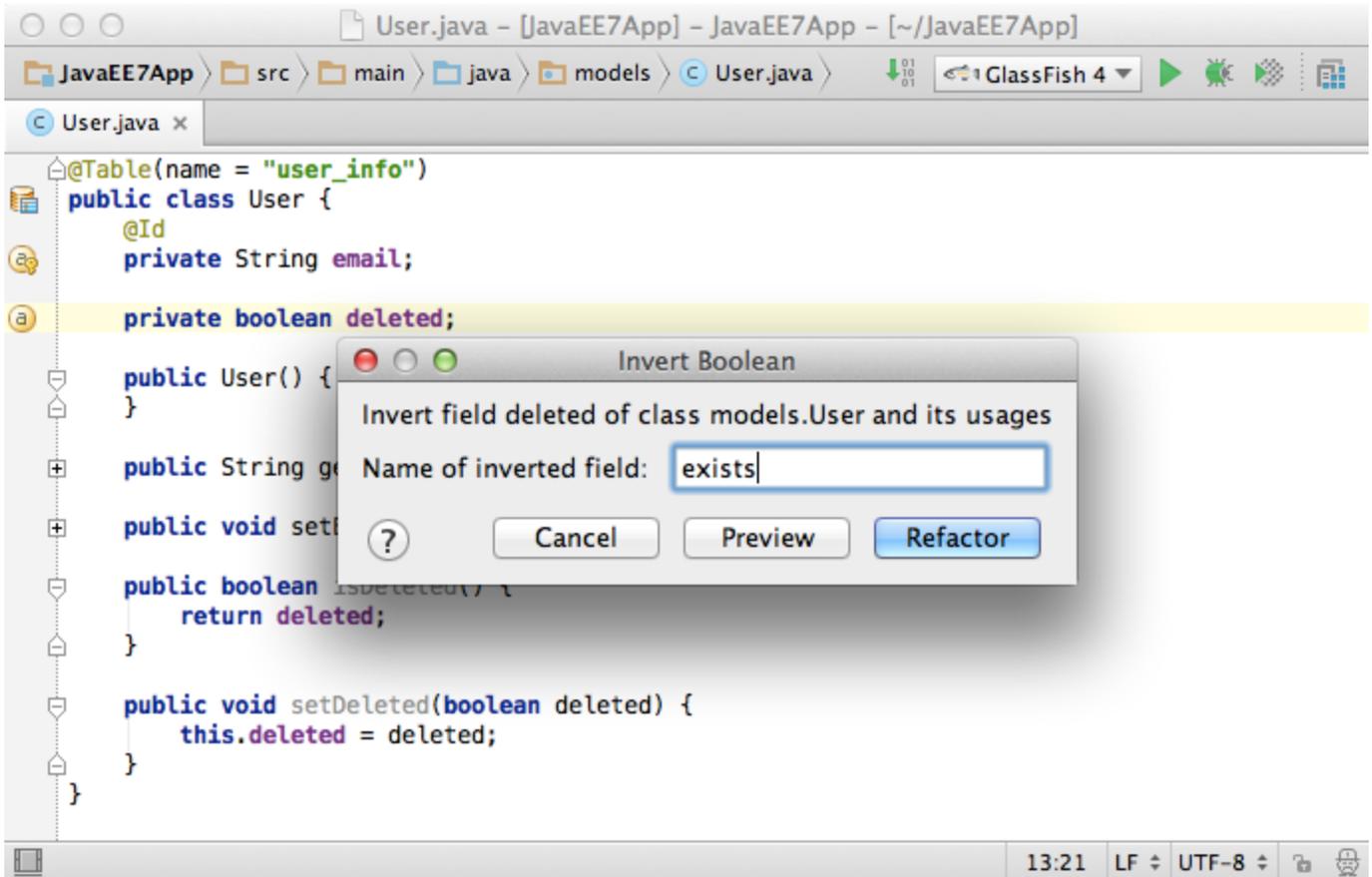
    @Before
    public void setUp() {
        user = new User();
        final List<String> emails = new ArrayList<String>();
    }

    @Test
    public void test() {
        Assert.assertEquals("demo@jetbrains.com", user.toString());
    }
}
```

The screenshot shows an IDE window titled "UserTest.java - [JavaEE7App] - JavaEE7App - [~/JavaEE7App]". The breadcrumb path is "JavaEE7App > src > test > java > models > UserTest.java". The code editor displays the source code for "UserTest.java". A tooltip is visible over the line "final List<String> emails = new ArrayList<String>();", with the text "Declare final" and a checked checkbox. The tooltip also lists several class options: "ArrayList<String>", "AbstractList<String>", "AbstractCollection<String>", "Object", and "Collection<String>". The status bar at the bottom shows "expected", "16:27/12", "LF", and "UTF-8".

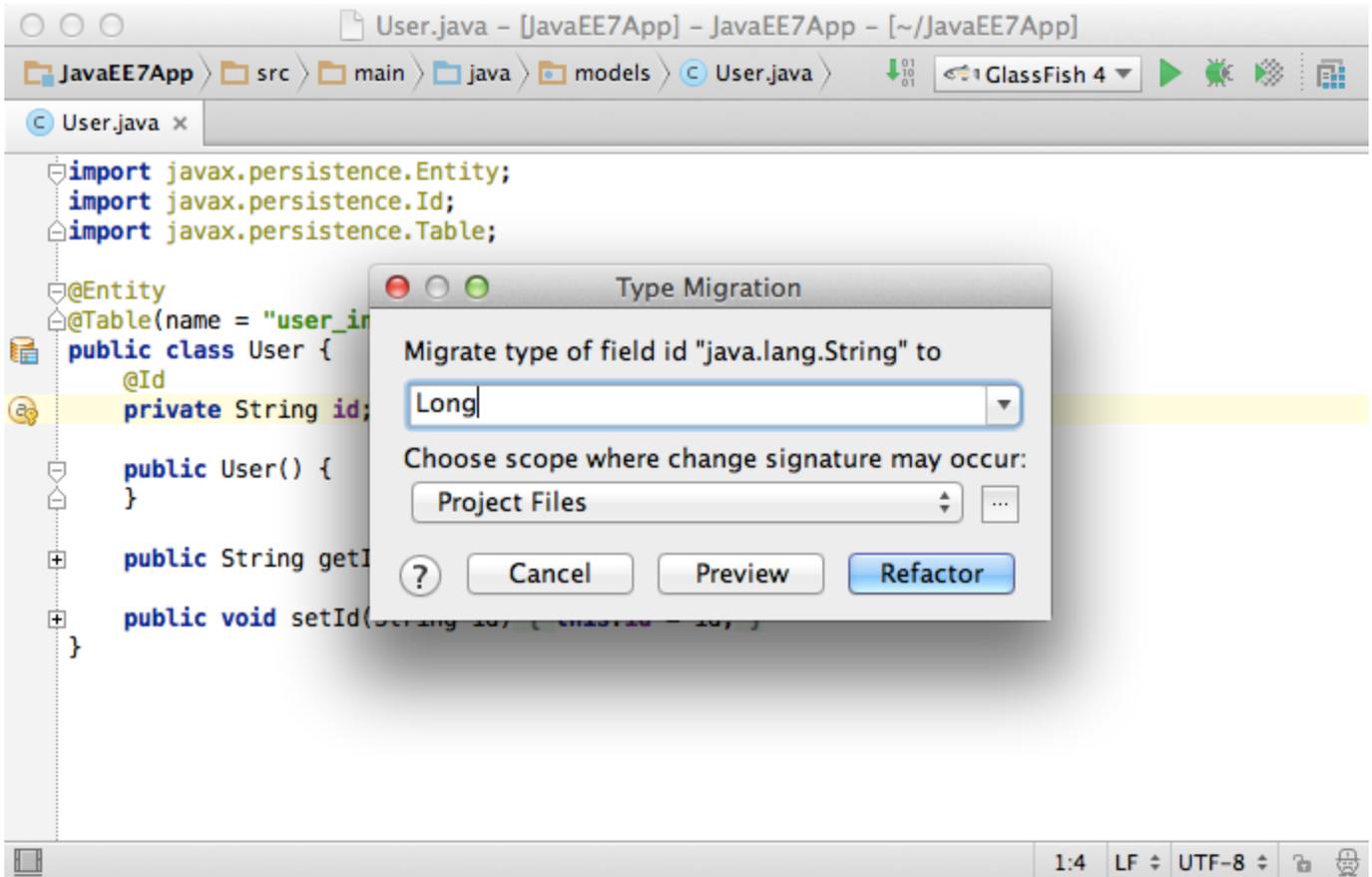
## 6. Invert boolean

One basic but very helpful refactoring is the Invert Boolean, which inverts the semantics along with the data flow dependent expressions for any Boolean variable, parameter, field or method.



## 7. Type migration

The Type Migration refactoring, as its name suggests, lets you automatically change the type for any class member, along with the dataflow-dependent type entries such as method return types, local variables, parameters, etc.



## 8. Essential refactoring shortcuts

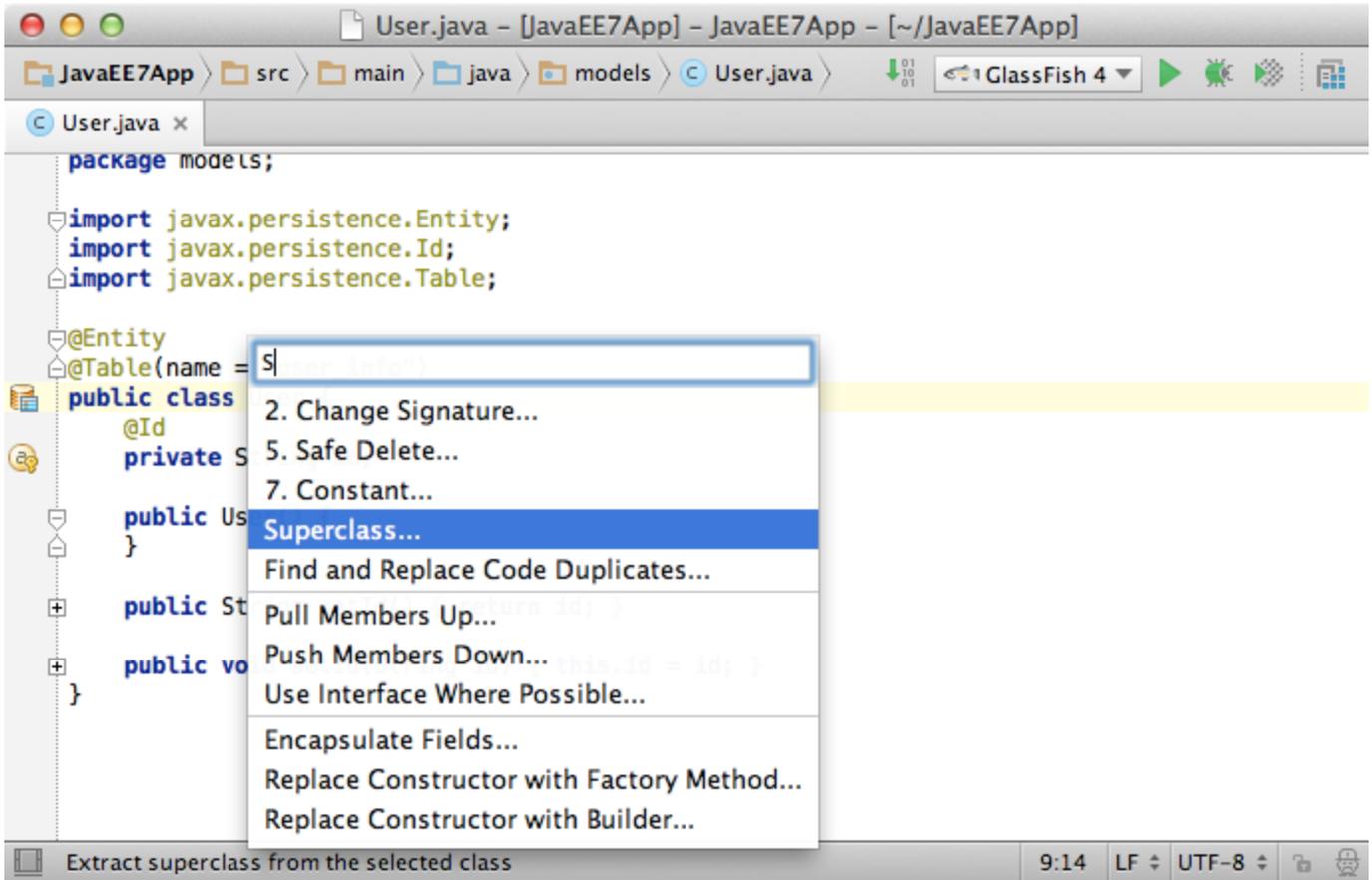
- Rename: Shift+F6
- Copy a class or file: F5
- Move a class or file: F6
- Extract a constant: Alt+Ctrl+C (Alt+Cmd+C for Mac)
- Extract a variable: Alt+Ctrl+V (Alt+Cmd+V for Mac)
- Extract a method: Alt+Ctrl+M (Alt+Cmd+M for Mac)
- Extract a field: Alt+Ctrl+F (Alt+Cmd+F for Mac)
- Extract a parameter: Alt+Ctrl+P (Alt+Cmd+P for Mac)
- Inline a class or method: Alt+Ctrl+N (Alt+Cmd+N for Mac)
- Change signature: Ctrl+F6 (Cmd+F6 for Mac)

## 9. Other useful refactorings

- Pull members up/down
- Extract a super class
- Extract an interface
- Convert an anonymous class to an inner class

## 10. Refactor this

If you cannot recall the shortcut for a particular refactoring, or you're just not sure what to do next, simply use 'Refactor this action' by pressing Ctrl+Shift+Alt+T (Cmd+Shift+Alt+T). You will see the list of refactorings available in the current context.



This is it for the refactoring basics. See the following tutorials for more advanced topics.

[Previous](#) | [Quick Popsups](#)

[Top](#) | [Quick Start](#)

[Next](#) | [Folding](#)