Dealing with Imports

Someday soon, someone will extend Eclipse with colorful graphics. Before you use an import handling action you'll see Barbara Eden coming out of a bottle.

"Jeannie, do anything that needs to be done with import declarations in my code."

"Yes, Master."

That's how Eclipse's Import actions work. You can ignore everything having to do with imports until the very last minute. Then choose Source-->Organize Imports or Source-->Add Import, and Eclipse performs its magic.

The Organize Imports action

Eclipse provides at least two ways to play with your code's import declarations. This section deals with the first way md choosing Source-->Organize Imports. Many things happen when you apply this Organize Imports action:

* **Eclipse removes any import declarations that you don't use.**

  If your code starts with

  ```
  import javax.swing.JButton;
  ```

  but you never use a JButton, then Eclipse deletes the JButton import declaration. Eclipse deletes the declaration even if you use JButton, but all of your references to JButton are fully qualified (as in `button = new javax.swing.JButton("Help")`).

* **Eclipse adds any missing import declarations.**

  If your code includes

  ```
  button = new JButton("Help");
  ```

  but you have no import declaration for JButton, Eclipse adds

  ```
  import javax.swing.JButton;
  ```

  near the top of your code. I've even seen Eclipse uncomment a declaration that I'd commented out earlier.

  What if your code refers to a mysterious Element type? The Java API has four different Element types, each in its own little package. Since Eclipse can't decide which Element type you want, Eclipse asks. (See Figure 1.)

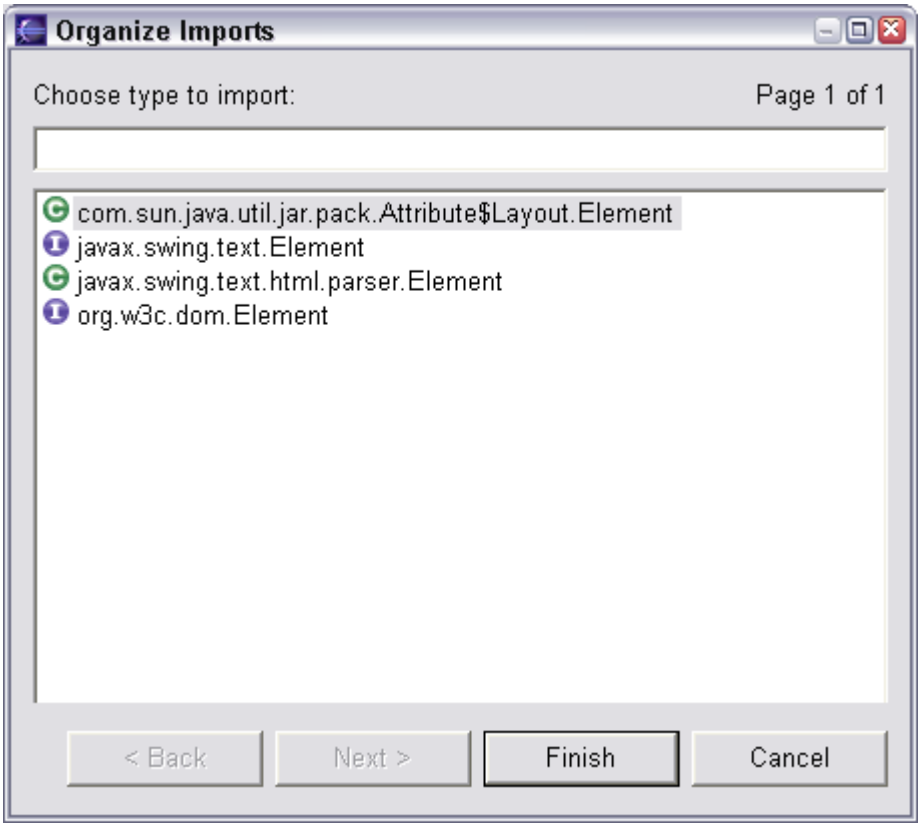

Figure 1: The Organize Imports wizard.

In Figure 1, you can start typing the name of your favorite package, or double-click one of the names in the list.

* **Eclipse sorts your code's import declarations.**

  By default, `java` packages come first, then come the `javax` packages, then the ~~`org`~~ packages, and finally the ~~`com`~~ packages. Within each category, Eclipse sorts declarations alphabetically. (That way, the declarations are easy to find.)

  Of course you can change the sorting order. Visit the Java-->Code Style -->Organize Imports page of the Window-->Preferences dialog. Move

names up in the list, move names down in the list, add names, or remove names. It's all up to you.

* **Eclipse tries to eliminate import-on-demand declarations.**

  If your code starts with

  ```
  import javax.swing.*;
  ```

  but you use only two Swing classes, Eclipse trades in the star for declarations like

  ```
  import javax.swing.JButton;
  import javax.swing.JLabel;
  ```

  Of course, if you have too many of these single-type import declarations, your code topples from its own weight. That's why you can configure the number of single-type declarations that Eclipse tolerates. (See Figure 1.) Personally, I think the default number 99 is too many. But who am I to say? I'm just an author.

The Add Import action

The Source-->Add Import action is a kind of mini Organize Imports command. But Add Import does things a bit differently:

* **Add Import acts on only one name at a time.**

  Place your cursor on a type name, and then choose Source-->Add Import. If only one package contains a class or interface with that name, Eclipse immediately adds an import declaration to your code. If more than one package contains a class or interface with that name, Eclipse offers you a choice of packages, as in Figure 1.

  Before you use Add Import this way, select as little source code as you can. If you select the entire line

  ```
  JButton button;
  ```

  then Add Import does nothing. If you select the word ~~button~~, on that same line, then Add Import still does nothing. But if you select only `JButton` (or just click your mouse anywhere inside the word `JButton`), then Add Import can create a new javax.swing.JButton import declaration.

* **Whereas Organize Imports removes unnecessary import declarations, Add Import can turn them into necessary import declarations.**

  Start with both of the lines

  ```
  import javax.swing.JButton;
  ```

  and

  ```
  panel.add(new javax.swing.JButton("Help"));
  ```

in your code, and make sure that you don't use the name JButton anywhere else.

* If you choose Source-->Organize Imports, then Eclipse removes the import declaration.

* If you select the `JButton` in `new javax.swing.JButton`, and then choose Source-->Add Import, Eclipse simplifies the constructor call. You end up with this statement:

```
panel.add(new JButton("Help"));
```

Add Import simplifies only one name at a time. So if your code contains the line

```
javax.swing.JButton button = new javax.swing.JButton();
```

then one application of Add Import can simplify either the first or the second `javax.swing.JButton`, but not both.

<Warning>

Don't apply Add Import when your cursor in on an import declaration. When I try it, Add Import turns `import javax.swing.JButton` into an invalid `import JButton` declaration.

<Warning>

Don't be fooled by the yellow highlight that you see when Eclipse marks occurrences. If you see a name that's highlighted yellow, don't assume that the name is selected for Add Import purposes.