

## How to build a plugin for Servoy – A step by step tutorial brought to you by Servoy Stuff

### PART 2

## F. The public API is your friend

In the first part of this tutorial we went through the tedious details of configuring our Servoy/Eclipse environment, and set-up our project “WhoisPlugin”, now it’s time to actually build the plugin.

But we need first to understand a bit more of the API interfaces we are going to use. The first one is the IClientPlugin interface and the second one is the IScriptObject interface.

Start by having a look at the javadocs of the IClientPlugin interface (refer to the javadocs online: <http://www.servoy.com/docs/public-api/> and choose the version that correspond the best to your target Servoy version – fortunately for these 2 interfaces, there isn’t any notable changes between each version– meaning our plugin should work well from Servoy 3.5.x to the latest version), click on the IClientPlugin link in the “All classes” frame, you will get this html page:

The screenshot displays the Javadoc for the `IClientPlugin` interface in Servoy 4.1.x. The browser window title is "IClientPlugin (Servoy 4.1.x Public API) - Windows Internet Explorer". The address bar shows the URL `http://www.servoy.com/docs/public-api/41x/index.html`. The page content includes a navigation menu with "Overview", "Package", "Class Tree", "Deprecated", and "Index". The main content area shows "com.servoy.j2db.plugins Interface IClientPlugin" and lists "All Superinterfaces: [EventListener](#), [IPlugin](#), com.servoy.j2db.persistence.ISupport...". The interface definition is shown as "public interface IClientPlugin extends [IPlugin](#), [PropertyChangeListener](#), com.servoy..." and is described as "Base interface for all client plugins." The left sidebar shows a list of "All Classes" including [HeadlessClientFactory](#), [IBeanManager](#), [IClientPlugin](#), [IClientPluginAccess](#), [ICmd](#), [ICmdManager](#), and [IComponent](#).

You can see that it states that this is the “Base interface for all client plugins.”

Just what we are looking for!

**Note:** For those of you not too fluent in Java (but who really want to know more), you should realize that an "interface" is basically a *contract*: you can build or use classes that will "*implement*" this contract, and you will be able to refer to any kind of objects by their interface regardless of what kind of object they truly are. This is known as "**Polymorphism**" and this is a very powerful and useful feature of Java (and most OO languages).

For example imagine that you have an interface of type Person with some properties like "name", "sex", whatever..., and some methods like "walk()", "talk()" etc..., now you can build (implement) lots of very different kind of person based on this interface, like a ServoyUser for instance or a TaxiDriver or a ServoyManager, and they each might have different methods attached... A ServoyUser might have a "writePluginForServoy()" method, when a TaxiDriver will have a "dontBehaveLikeRobertDeNiro()" method and a ServoyManager could have a "tryToIgnorePeopleSayingTheServoyDocNeedsUpdating()" method....

To be valid "People" in any case, these classes will need to *implement* the methods that you have declared in the People interface. In fact, the java compiler will enforce that!

Now the good part is that, depending on what you want to do with these objects, you might prefer considering them as simple "People", in which case you will get access to the "name" and "sex" properties and to the "walk()" and "talk()" methods regardless of whether you are using a ServoyUser a TaxiDriver or a ServoyManager object (but if you refer to them as People you will not have access to their "specialized methods", unless you perform a "cast".

You can Google "java interface" for more information on this (overly discussed) subject...

So now that we know that the IClientPlugin interface is the "Base interface for all client plugins" we are going to *implement* it.

\*"*implement*" really means to "code" the behaviour, but you should get used to the term, since it is more precise and it is used widely when referring to the fact that you "fulfill the contract" of an interface, in fact you will often find classes named something like "MyInterface**Impl**" – this is to indicate that it is a "class that implements the MyInterface contract".

## G. The IClientPlugin interface

Now, let's have a look at the methods that we need to code to "fulfill the contract" in the javadocs (javadocs are nothing but simple yet effective html documentation built automatically - yes automatically! - from special comments that you insert in your code), you can find them in the table called "Method Summary":

```
public interface IClientPlugin
extends IPlugin, PropertyChangeListener, com.servoy.j2db.persistence.ISupportName
```

Base interface for all client plugins.

---

## Field Summary

Fields inherited from interface [com.servoy.j2db.plugins.IPlugin](#)

[DISPLAY\\_NAME](#)

---

## Method Summary

<a href="#">Icon</a>	<a href="#">getImage ()</a> Get the plugin image (16x16 px), used in the developer debugger treeview and preference tabpanel.
<a href="#">String</a>	<a href="#">getName ()</a> returns the (JavaScript)name for the plugin,name SHOULD apply to the JAVA identifier rules.
<a href="#">PreferencePanel</a> []	<a href="#">getPreferencePanels ()</a> Create panels for the Preferences dialog, lazy called when shown.
<a href="#">IScriptObject</a>	<a href="#">getScriptObject ()</a> returns the object which has 'js_xxxx' methods for script calling, or null if not scriptable
void	<a href="#">initialize (IClientPluginAccess app)</a> Called on startup after client application started.

---

Methods inherited from interface [com.servoy.j2db.plugins.IPlugin](#)

[getProperties](#), [load](#), [unload](#)

---

Methods inherited from interface [java.beans.PropertyChangeListener](#)

[propertyChange](#)

The name of the methods is pretty self-explanatory:

- `getImage()` should return an `Icon` - this is the one you will find in the Solution Explorer in developer under the plugins node for your plugin.
- `getName()` should return a `String` – of course it will tell Servoy the name of the plugin (the one you will see in the developer node and the one you will use in Servoy with the syntax `plugins.nameOfMyPlugin.something()`)
- `getPreferencePanels()` should return an array of “`PreferencePanel`” objects, this one is a lot less used, basically you only need to use it if you want to add a panel of configurable options in the “Application Preferences” dialog of the Servoy Smart client. We are not going to use it today (maybe for another plugin in another tutorial), if you are very curious about this and want to see an example, look at the sources of the Agent plugin included in the `/plugins/agent.jar` file.
- `getScriptObject()` should return an object of type `IScriptObject` – aha! That’s the one which will do all the scripting work for you when you use scripting in Servoy: the core object of you plugin, so we will look at it in more details since we will implement it.
- `initialize()` returns nothing, this time Servoy will call that method for you, giving you access to the client by way of an `IClientPluginAccess` object parameter (we will look at this one later).

But wait! There is more, if you look carefully (because you do if you are a serious java programmer), you will see that the IClientPlugin interface also inherited\* from the IPlugin interface (a more general interface than that one), this IPlugin interface has defined 3 methods:

- `getProperties()` should return a Properties file (this is a standard java object, usually it is a set of keys/values pair, you often find them as ".properties" files with prefix.key=value lines), this is where Servoy will get some more information on your plugin, but most of the time all you will have to give it is the "DISPLAY\_NAME" value – which believe it or not, will be the name of your plugin.
- `load()` is a method that Servoy will call on your plugin when the application starts, apparently – not sure exactly when this is in the sequence of events in the lifecycle of a Servoy application – this is where some in-depth sequence diagram would sure help, does someone in Servoy is listening?
- `unload()` is a method that Servoy will call on your plugin when it will get rid of it (apparently "on application shutdown" – see above for some kind of comments regarding the obscurity of that event in the Servoy lifecycle). Anyway, this is where you will dispose of the objects that you created or kept in reference, to avoid memory leak.

And finally, you should note that the IClientPlugin interface is also inheriting\* from PropertyChangeListener, meaning that it might be called on changes of some bound properties with a call to the (only) method of the interface:

- `propertyChange()` will be called by Servoy when some bound properties have changed, you will get a PropertyChangeEvent in parameters with the option to act on it. These bound properties are the one you will use in your "PreferencePanels" if you have created any. Again, see the Agent plugin to see how this can be used.

\* Interfaces can *inherit* from other interfaces, building a kind of hierarchy of interfaces which will add some specialization to the broader use of the parent interface.

In our "People" interface example, we could have designed a "ServoyPeople" interface "extending" the "People" interface, meaning that it would have added some specialized methods to it: a ServoyPeople for example would have defined a "discussTheUrgencyOfUpdatingTheServoyDoc()" method, and each derived objects like ServoyUser and ServoyManager would have had to implement that method (although in some very different way ;-).

## H. Implementing the IClientPlugin interface

Now is the time to create our plugin by implementing the IClientPlugin interface first.

To do this, we will create a class and name it after our plugin (this is not required but it's best to name the class after what it is supposed to represent!). And to follow the best practices we will place our classes into a nice Java "package"\*.

\* A **package** is a neat way in Java to organize libraries coming from arbitrary sources and make them work together without problems of "Name collision". It is basically a hierarchy of folders in your sources that will isolate your classes from those of others that might have the same name (for those of you who are XML gurus, this is very similar to namespaces).

To better understand Java packages, think of this silly example:

Imagine that there is two java developer wannabe who wrote a clever Date class.

Imagine that they didn't use any java package to put them into.

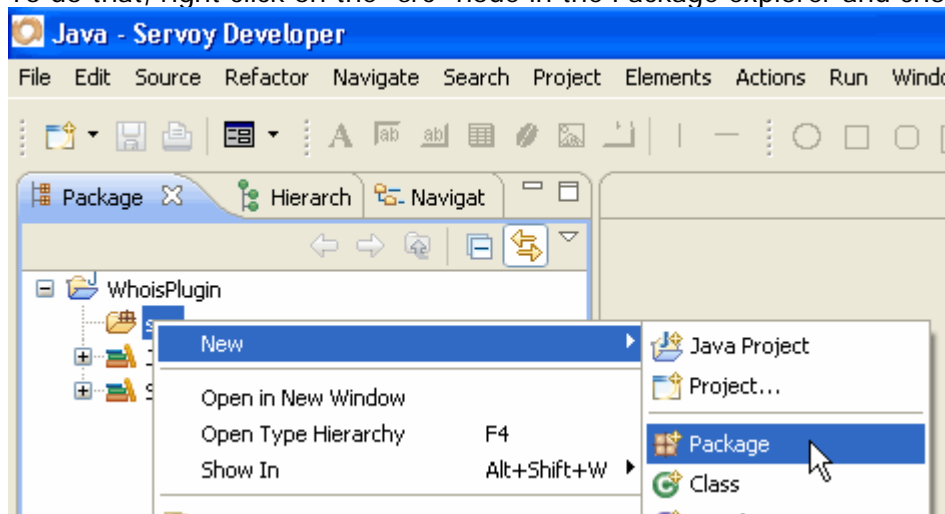
Now comes a third developer, building a java application (a java stack) using different utility libraries, and he integrates both of the Date libraries inside his application (maybe even without knowing it, because they might be used by some other libraries themselves).

Now when he wants to use a Date class which one is it going to be? The compiler will not know, and in Java, when the compiler doesn't know which class is which, that means trouble and generally some sort "ClassCastException" or "ClassNotFoundException" or some other niceties...

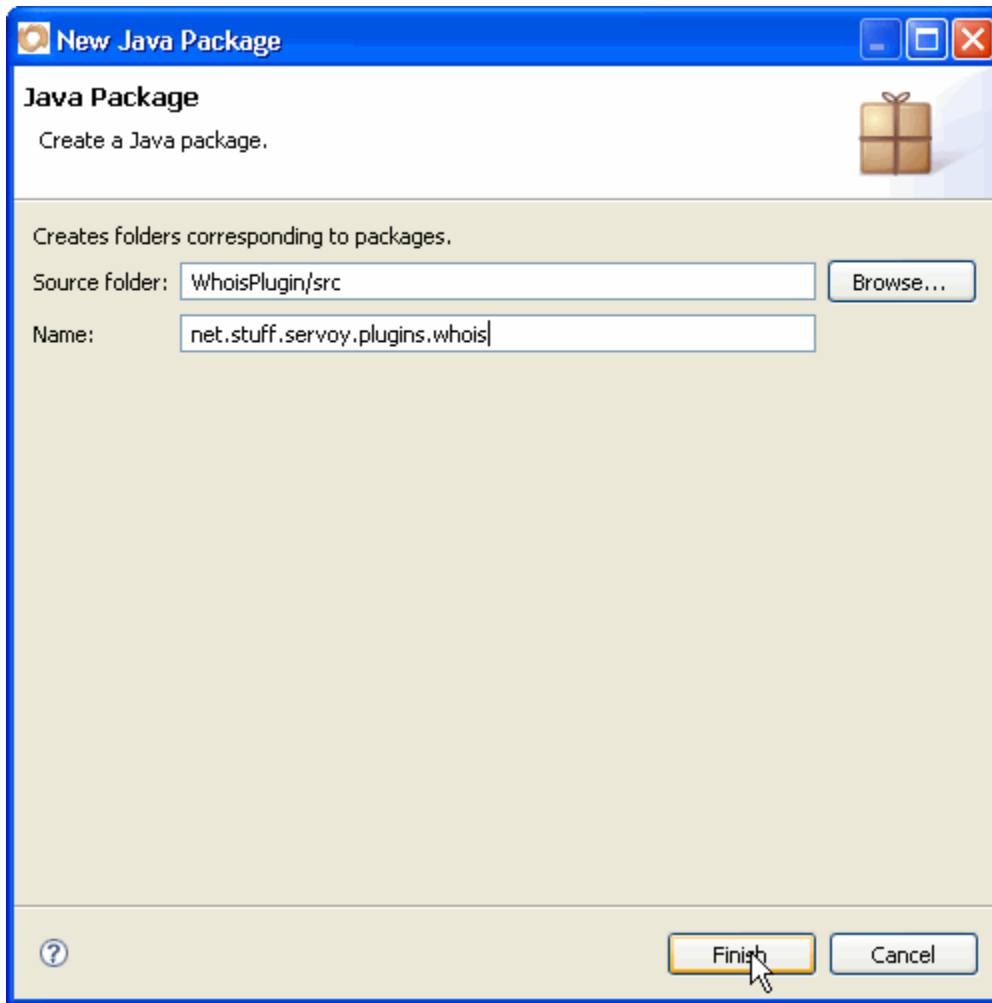
This is where java packages come to the rescue, by building a hierarchy of folder and using "**fully qualified names**" (full means prefixed with their packages) to access our classes the compiler will know which one is which.

There will be a "com.wannabe.first.Date" class and a "net.wannabe.second.Date" class, two totally different classes, you see?

So, we will start by adding some "packaging" to our sources and we will then add classes to them. To do that, right click on the "src" node in the Package explorer and choose "New > Package":



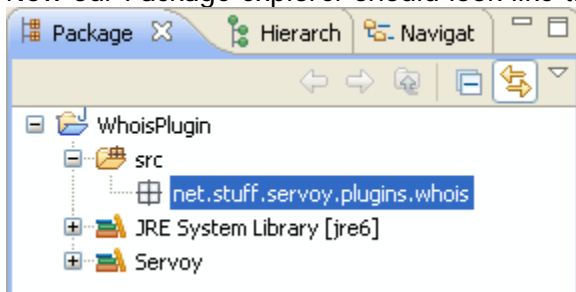
Now, let's give our package some name in the following window:



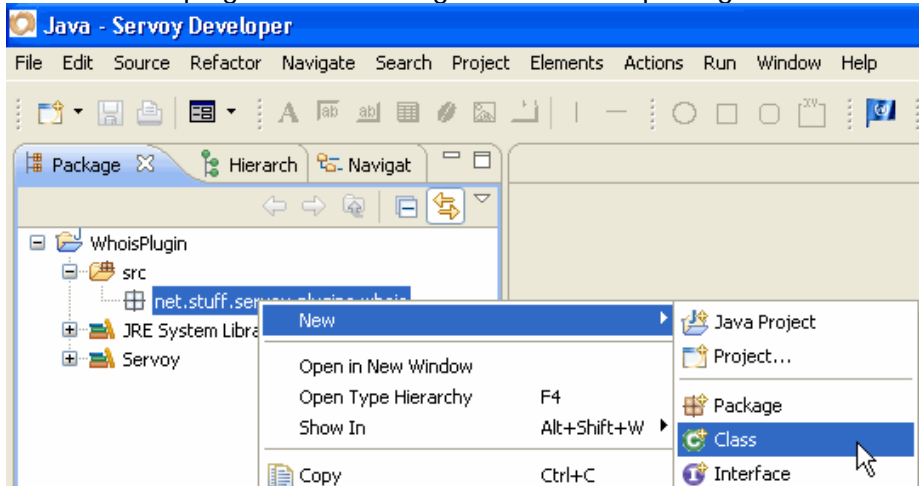
**Note:** You can name it whatever you want but, by convention, the package hierarchy should reflect your domain name (if you have any) in reverse (from domain to sub-domains), so for example, I prefix all the development I do for Servoy Stuff with net.stuff.servoy (because my web site is servoy-stuff.net) – each dot “.” being a separator will create a new folder in the “src” folder. Then I follow with the type of thing this is (plugins or beans) and then the name of the stuff itself.

You might think that this is an awfully long name, but this is pretty standard package name: it will avoid for sure any name collision with any other libraries, and anyway you will probably never have to type the “fully qualified name” in your code, Eclipse will automatically do that for you (by adding the package name on top of your java file and by adding the relevant “import” with the “fully qualified package name” for the classes you use).

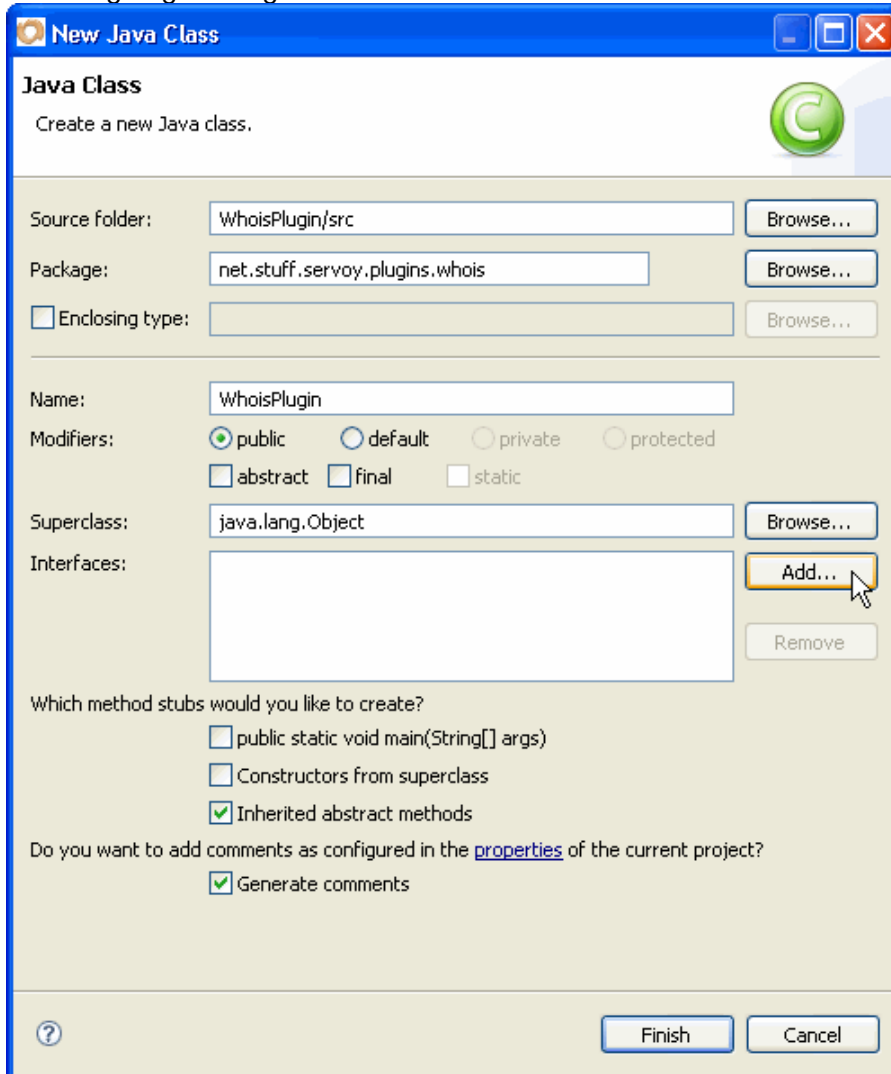
Now our Package explorer should look like this:



Let's add our plugin class now: right-click on the package and choose "New > Class"



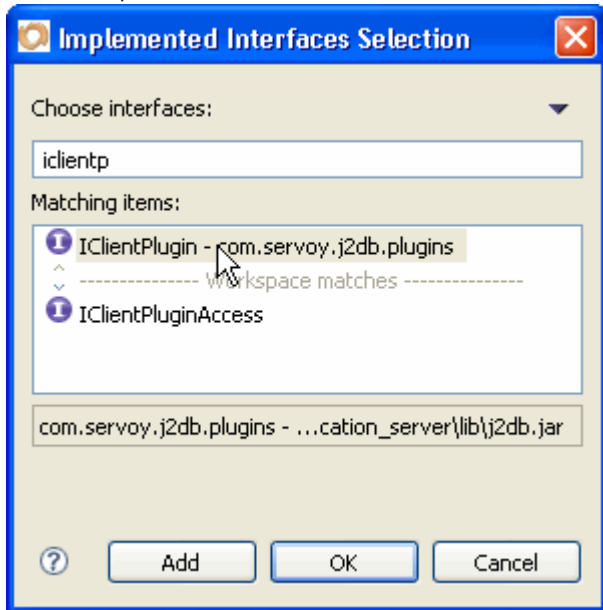
This brings the "New class" dialog window, believe it or not, this is where most of the code in your class is going to be generated:



**Note** that the "Source folder" is already filled with the correct location of our sources "WhoisPlugin/src" (in our workspace); **note** also that the "Package" is the one we have selected.

Now we will type the name of our plugin "WhoisPlugin", - keep the Modifiers as is, our class will need to be public so that Servoy will find it, it will not be abstract because it will concretely implement something and it will not be final because you never know when the time will come to extend it. The superclass (the "parent" of our class) will only need to be "java.lang.Object" (there will be time when you will have a whole library of "generic plugin" classes that will serve as parent objects for your new plugins but this isn't the case yet ;-)

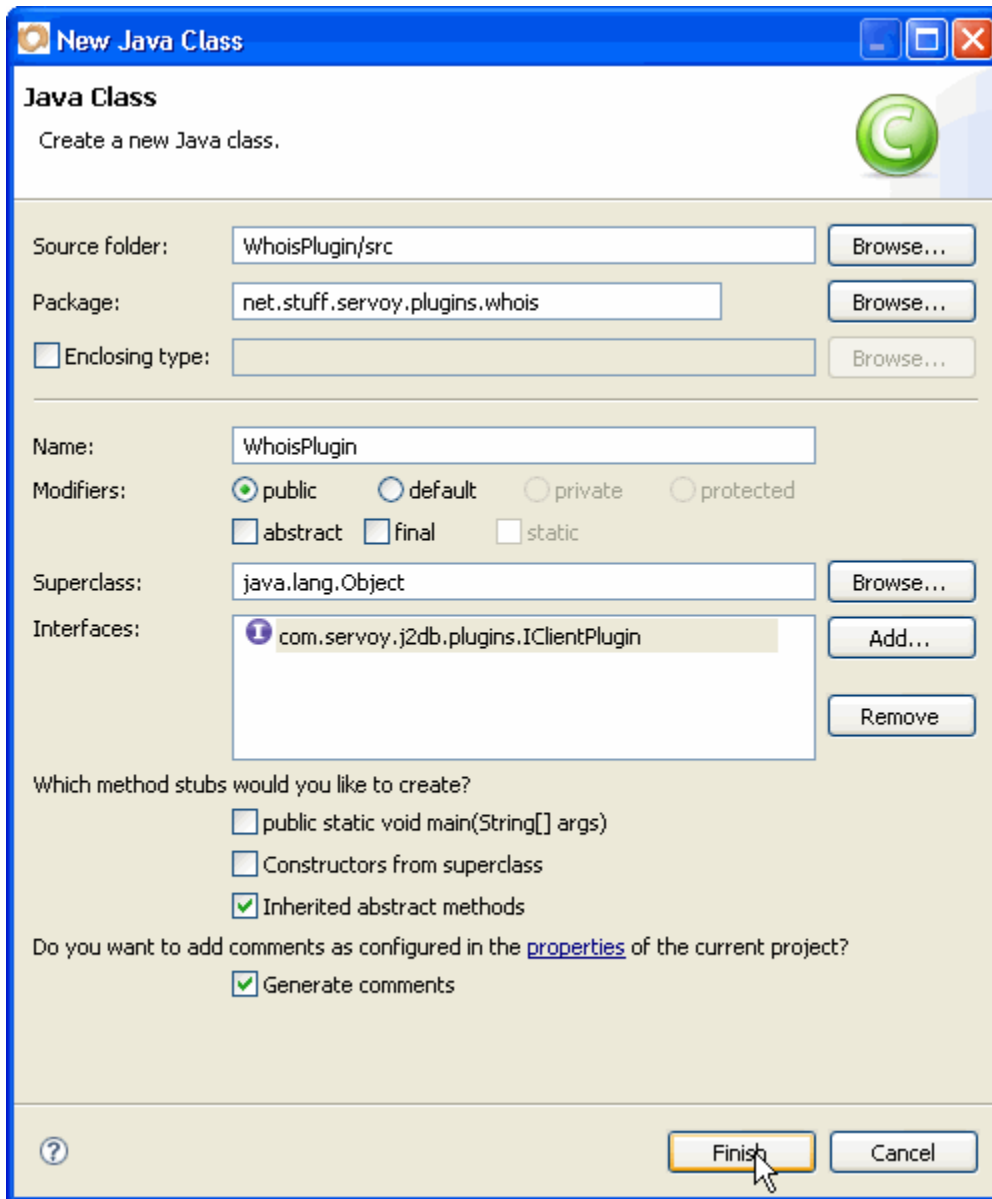
What is really important here is to tell Eclipse that we want our new class to implement to IClientPlugin interface, so we click the "Add..." button in front of the "Interfaces:" list area.



All you have to do here in this clever little dialog box is to type the first letter of the interface you're looking for and Eclipse will show you "as you type it" the found interfaces (it is clever to a point where the next time you open this window it will remember what you did before and will show you the last chosen interfaces).

In the text box, type the first letters of IClientPlugin and you will quickly see the line "IClientPlugin - com.servoy.j2db.plugins" appear in the "Matching Items" box. Double click on this line and click "OK", the interface should have been added to the previous dialog, like that:





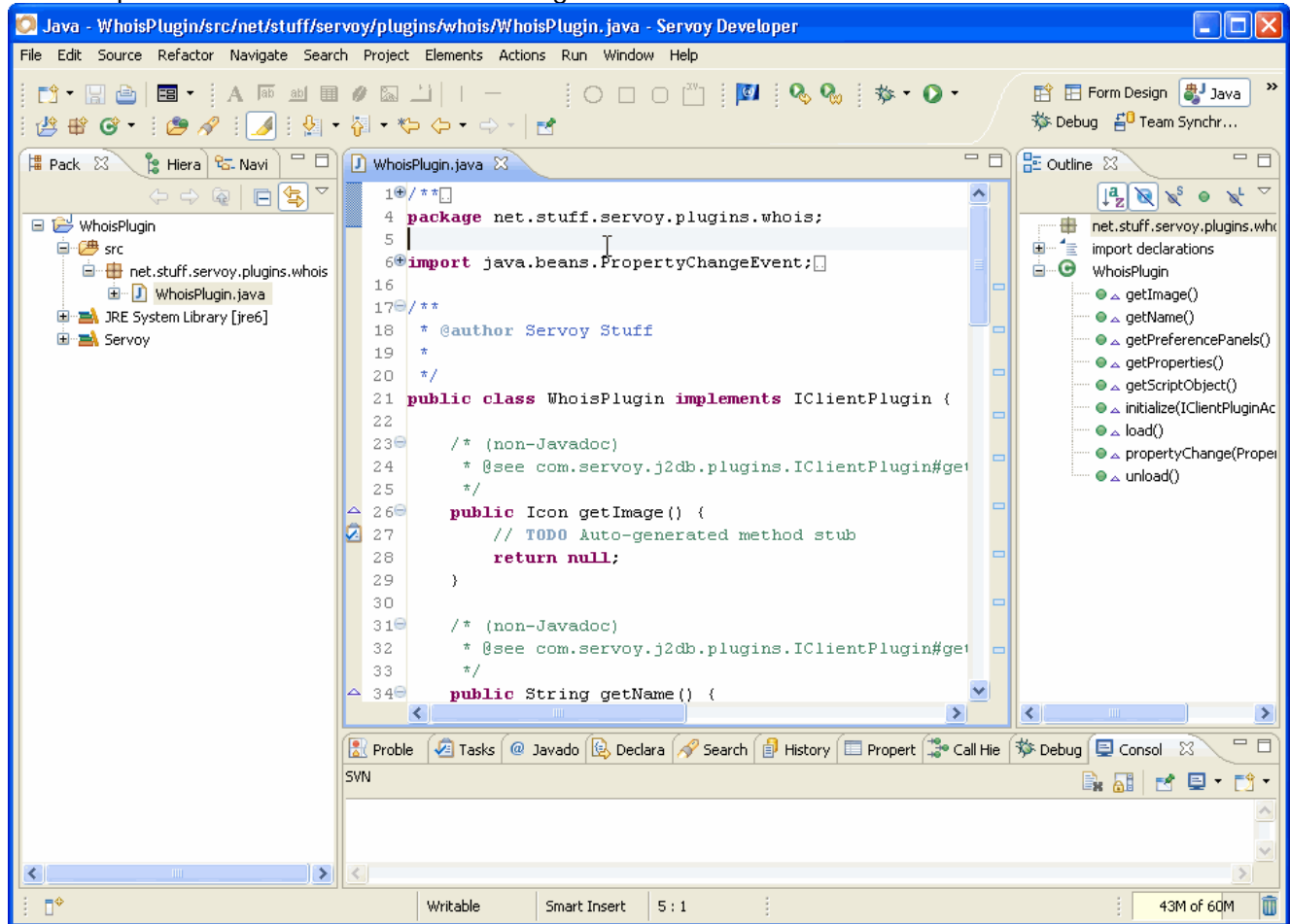
We don't want our class to be an application, so no need for a main() method, don't check that. We don't care about constructors from superclass (the java.lang.Object default constructor will do), so leave that unchecked.

We DO want to inherit abstract methods because then Eclipse will generate the "method stubs" from the interface(s) we selected.

And it's nice of him to propose to generate comments, so we check that too - the way it will generate comments really depends on the state of your Eclipse installation, it can be totally customized – follow the link "properties" to see how if you want.

We finally click the "Finish" button to get the resulting WhoisPlugin class generated for us, placed in the right package and opened in the java editor, we did it! Well almost.

Your Eclipse IDE should now look something like this:



Now there is a lot to see in this screen, there is a lot to talk about to reassure those of you who glare at this kind of screen for the first time. You should believe me that everything here is laid out for your convenience and to assist you in your task, everything is here only to help, not get in your way!

First **note** that the “WhoisPlugin.java” source is already filled with code, meaning that we will only have to type a minimal amount of code to implement our plugin.

In the “Package” explorer, the class we created is now in the correct package. And depending on whether your Outline view is opened (the right panel); you can see a tree of all the methods that have been inserted.

Depending on the preferences set up for your comments they might be there or not, but as you can see on the capture above anyway, there are some strange “/\* (non-Javadoc)” comments – they are here to tell the javadocs system (remember the system that can generate automatic html doc from the comments in the java source file?) that the comment is to be find in a superclass or an interface, and there is also a reference to the relevant superclass or interface method in the special tag “@see”.

You can also see that since we created our class in the “net.stuff.servoy.plugins.whois” (or whatever you chose to name yours) package, the first line of code is declaring it.

Then there are some lines of code for the “imports”, - these are there so that you won’t need to type fully-qualified names each time you use a class, they allow you to use a syntax like “new Date()” and if

the "import java.util.Date;" or "import java.util.\*" is here on top of the file, the compiler will know precisely which Date class you are referring to.

**Note** also the little + or – sign in front of some lines, this indicates that some part of the code is wrapped, (in which case you can unwrap it with the + sign) or unwrap (in which case you can wrap it with the – sign) – you get the same one in the JavaScript editor in Servoy.

This is **Eclipse unified interface** to you, and in fact you know more about the Java editor than you think since it is based on the same rules as the JavaScript editor your use in Servoy, only this time adapted/extended for Java, as we will see later...

Now the declaration of the class is the line:

```
public class WhoisPlugin implements IClientPlugin {
```

- of course you will have a corresponding close bracket "}" at the end of the file to close the class definition)

This is the correct syntax to tell the compiler that your class will be accessible from outside of your library (**public**), that its name is WhoisPlugin (and in that case the name of the file will have to be "WhoisPlugin.java", and will be compile as "WhoisPlugin.class") and that it *implements* the IClientPlugin interface, meaning that you (and others) will be able to use it exactly as any other class implementing this interface regardless of what's inside, which is exactly what Servoy is going to do, considering only its interface (and thus the set of methods that this interface declare) and nothing else that could be programmed inside, thus using your class as a **Plugin!**

Ignore the "/\* (non-javadocs)" comments and you get right to the first method:

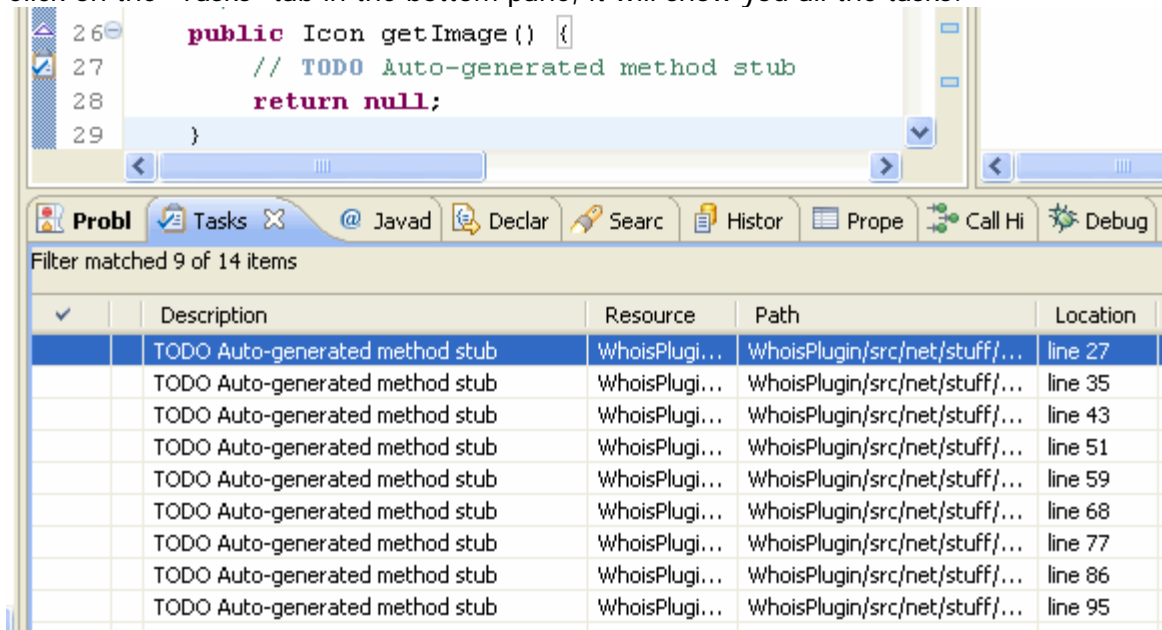
```
public Icon getImage() {  
    // TODO Auto-generated method stub  
    return null;  
}
```

**Note** the single line comment starting with "// **TODO**"

This time this is Eclipse telling you what you have to do!

And a nice thing about this is that you can view all the corresponding tags, called "TASKS" in the "Tasks" view, and see instantly what you have left to do.

Click on the “Tasks” tab in the bottom pane, it will show you all the tasks:



You see that you have quite a few things to do to, and this panel will show them all, and tell you in which file it is, and precisely at what line it is. You can double-click on any of these lines and it will go straight to it, opening the file if necessary.

**Note** that you can add your own TODO tasks anywhere in your own sources, to trace what's left to do, it happens when you are doing a complicated task that you will break in lots of little one, and each time you leave a blank, its good to place one of these tasks markers so not to forget them after that. All you have to do is create a single line comment `//` and type `TODO` in capital letters, as soon as you save the file, the task will be added to the Tasks view.

**Note** also that there are other tasks markers (by default there is `FIXME` – with a highest priority – and `XXX` – undefined task) and that you can also create your own in the “Preferences/Java/Compiler/Task tags” dialog – refer to Eclipse doc to know more about tasks.

For example, I usually add a `CURRENT` tag, and a `DONE` tag, just to keep trace of things in my sources...

So now, you know precisely what's left to do: replace all the `TODO` tasks in the file by proper code.

## I. One method at the time

### 1. `getImage()`

First method, `getImage()`, its “signature” (the way it is declared) says that we must return an object of type “Icon”. If you guessed that it's going to be the icon of you plugin, the one that will show in the Solution Explorer of Servoy under the plugins node, you guessed right.

But what kind of object is an `Icon`? If you have doubts, you can first try moving your mouse over the `Icon` word; it will show you a little tooltip with a definition, coming straight from the javadocs system:

```
public Icon getImage() {  
    //  
    ret  
}  
  
/* (non  
 * @see  
 */  
public  
    //  
    ret
```

**javax.swing.Icon**  
A small fixed size picture, typically used to decorate components.  
**See Also:**  
ImageIcon

So now, you know that the Icon is more precisely a javax.swing.Icon (a standard Swing object then). It is in fact an interface, and you will be able to find a wealth of examples and usage on the internet by googling "javax.swing.Icon", starting with the full javadocs of the interface (that you can find here: <http://java.sun.com/j2se/1.5.0/docs/api/javax/swing/Icon.html>) – remember, the API is your friend! There are lots of information (usually) in the javadocs, especially the one from the standard Java language (unfortunately not the one from Servoy), and it's quite easy to find ways of using it.

For example, if you follow the "ImageIcon" link of the see also part of the Icon javadocs, you will see one of the standard classes implementing the Icon interface, and in the javadocs of this class there is a link to the "How to Use Icons" tutorials (I told you the API is your friend!).

I'm not going to tell you all the possible ways to create an Icon object, but we are going to use a fairly standard way (for a library), which is to use resources from the project itself.

First, we need an image (a gif, jpeg or png image) of a maximum size of 16x16 pixels (this is an icon, not a poster!), so get your best loved image editor, and create it, or go to the internet and find some open source icon resources to choose one. In case you are definitively too lazy to make your own or even search for one, you will find it in the source of this project.

But beware! I'm no artist, so here it is:



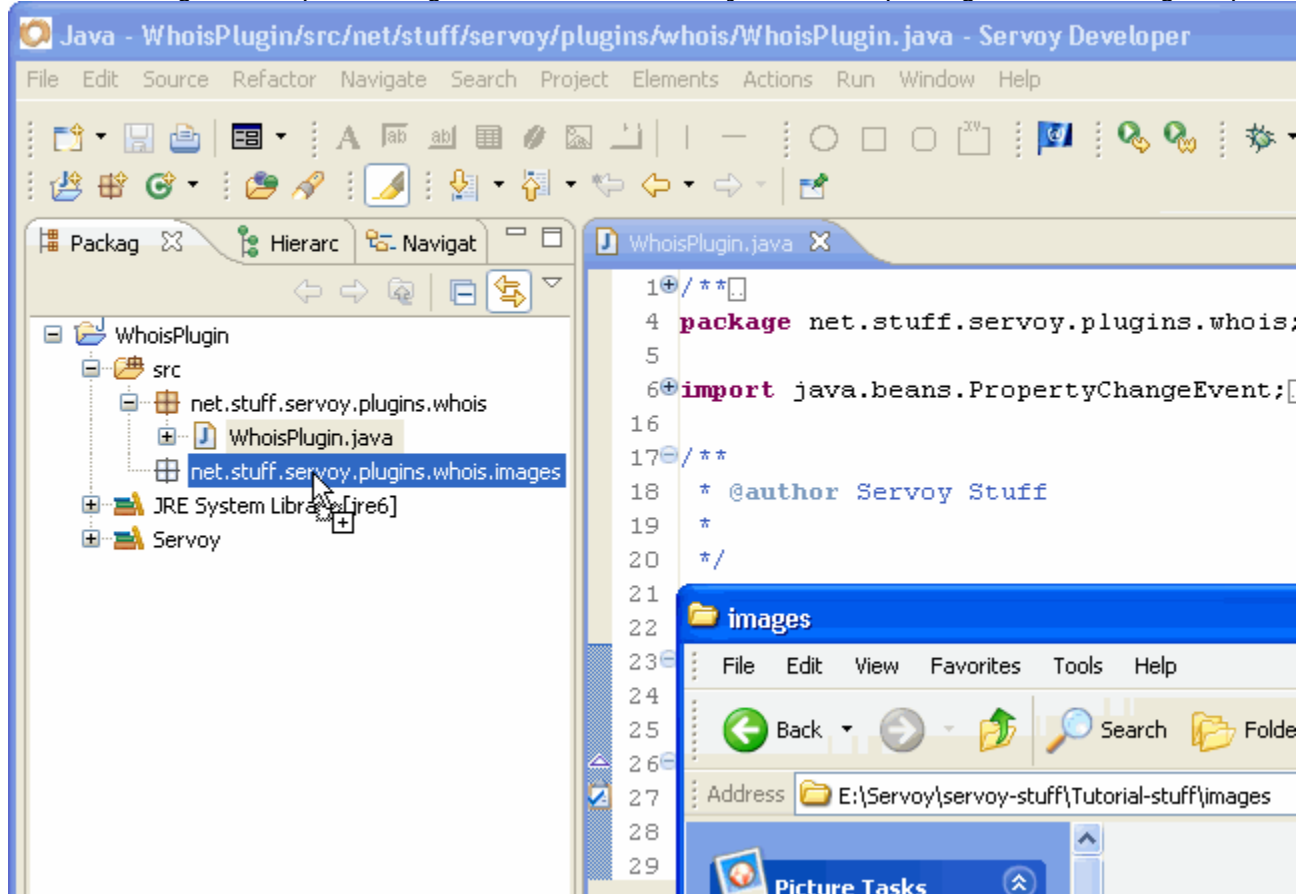
Told you ;-)

You can download it here: <http://www.servoy-stuff.net/tutorials/utils/t01/whois.png>

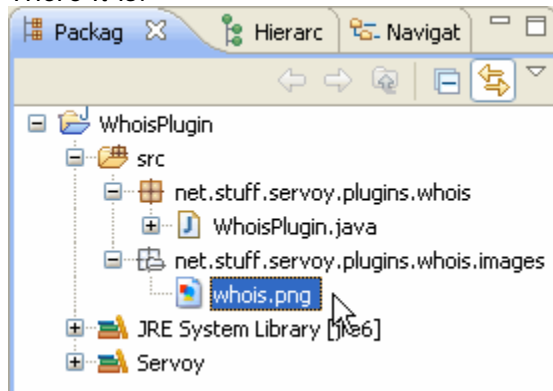
Now we've got an image, let's place it in our project in our package, or even better in a special "folder" inside our package (to keep things tidier©)

So let's create a package by right-clicking on the "net.stuff.servoy.plugins.whois", choosing "New > Package" and typing: net.stuff.servoy.plugins.whois.images

Now let's drag and drop our image from our disk directly inside the package in the Package explorer:



There it is:



And from there are we going to load it, because once compiled and deployed, it will be part of our "jar" file. If you open most of the standard Servoy plugin (the one having customized icon), you will see that the images are there too, inside the sources packages. The big advantage of this is that the image will be deployed at the same time and in the same place as your plugin classes, so you classes will always be able to find the image.

How do we load the image, then?

Remember that you cannot create an interface, it is not a class, it is not an object - it is a *contract*. Thus we need a class that implements this interface, and sure enough there is a standard Swing class that implements this interface as we learned from the java API: the ImageIcon class.

Now, if you look at the constructors of the ImageIcon class (in the API), you can see that you can give it a URL. How do we get the URL of a file that is inside our sources?

The easy way is to use the Class object of our class. The Class object has a method which will give us a URL based on a path String relative to its own location, so the syntax will be:

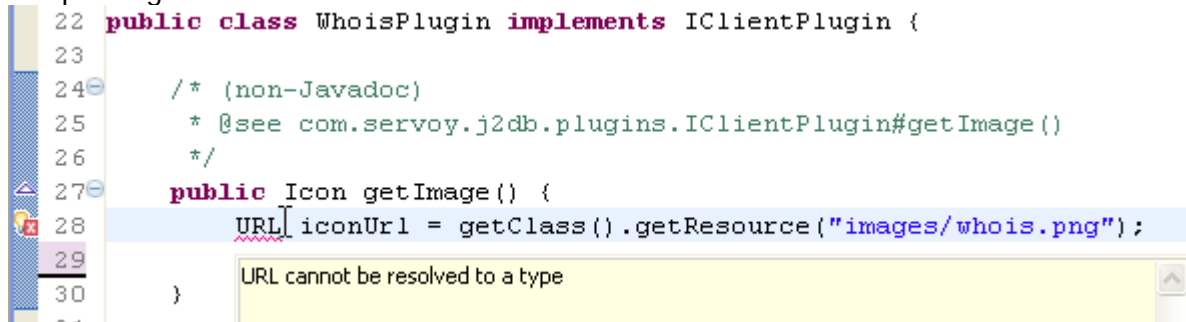
```
URL iconUrl = getClass().getResource("images/whois.png");
```

Where URL is really java.net.URL, and getClass() is a method from the Object class giving you access to the class of the object. It can also be written:

```
this.getClass().getResource("images/whois.png");
```

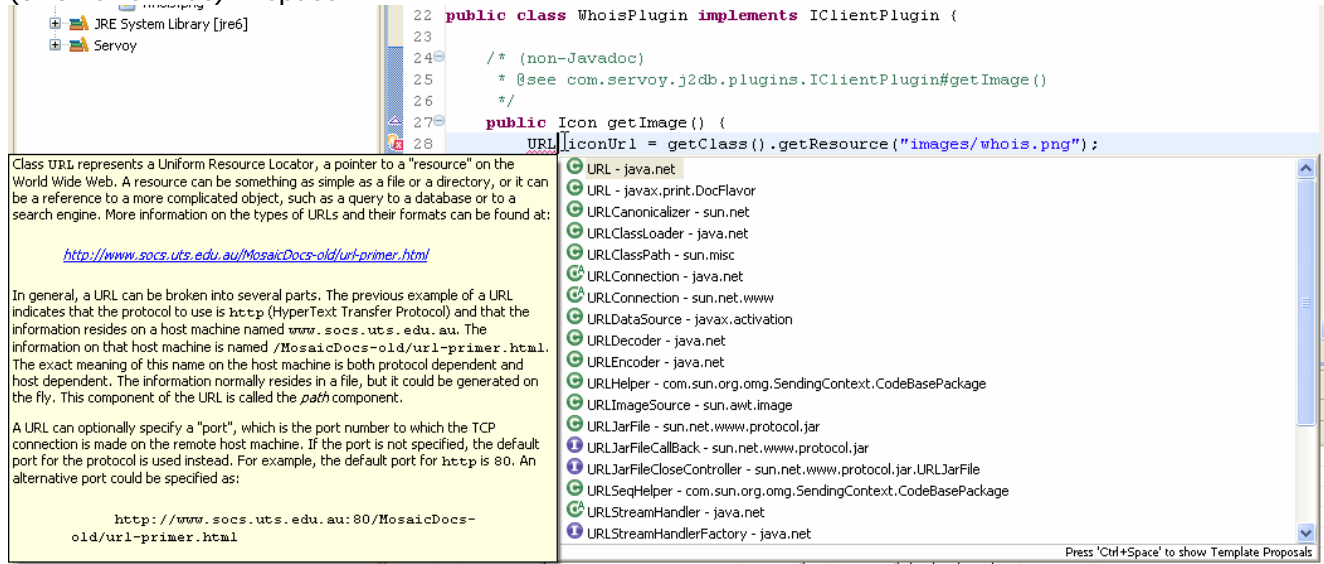
The "this" keyword refers to the object itself, and in that case it is not even necessary because there is no possible ambiguity.

Now if you copy this code or if you type it directly in the java editor, you will see that Eclipse is complaining about the fact that it doesn't know about URL:



There is a little squiggly line under the URL word, and if you hover your mouse over it, you will see that "URL cannot be resolved to a type". Notice also that on the right of the line, just next to the line number there is a little icon with an x on red background telling you that something is wrong, this is known as the "quick fix" icon, and it is one of the more powerful features of the java editor in Eclipse.

How to resolve the type then? Two ways: either place your cursor just after the URL word and type ctrl (or cmd for Mac) + space:



You will be able to choose the appropriate class from the list (and note that it will show you the javadocs for the selected class to help you finding the appropriate one).

Right now the URL we want is really the java.net.URL, so choose this one on the list, what Eclipse is going to do is to add the line "import java.net.URL;" on top of your file, and this time your URL will be recognized and your java file will be able to compile into a class file.

You can also click the "quick fix" button; you will get this kind of choice:



This time you will get more choice, like adding some import, but also to directly create a new class definition file or interface file or changing to some close know class names (useful when you made a typo).

If you chose this option to fix the problem, choose the second "Import 'URL' (java.net)"

Keep an eye on the "quick fix" button, when it appears, when you click on it (or right-click for even more options), it will help you fix quickly a lot of problems, it's like having a more experienced java coder right under your mouse!

So now our line is correct, and we just created an URL from a related path, now if it was found, let's use it to create an ImageIcon, so the rest of the code will be:

```

if (iconUrl != null) {
    return new ImageIcon(iconUrl);
} else {
    return null;
}

```

And we will have the same problem because our ImageIcon word will not be recognized as a class, so I leave you to resolve this problem, since you know (you're the developer after all) that it really is a javax.swing.ImageIcon).

But wait! Why did we check that iconUrl is not null before creating our ImageIcon?

Well, you should know that java code (as most computer code in fact) is governed by Murphy's Law: if something can go wrong, it WILL go wrong. So each time you type something freely (meaning not checked by the compiler - and that's the case each time you type a String constant), you're better off checking for something that might go wrong: you might have done a typo, you might later want to rename your "whois.png" file, your code will change...

If the code that changed was the name of a class or variable, the compiler will automatically detect that there is a problem and would refuse to compile your class, and this is good!

This will give you the opportunity to do something before the bug hit your users!



But the compiler will not be able to check what's inside your Strings, so if the day you change the name of you image you completely forgotten that you used it in this class, the compiler will not complain, but you user sure will, because it will cause a well know and beloved java NPE (short for NullPointerException) and your code will not work at all after that!

It will be like that because if you pass a null URL to the constructor of the ImageIcon class, boom, NPE! But if you are a clever java developer and you know the drill, you will have checked for that kind of future bug in advance and prepared a counter-offensive: in that case, if for some unknown (yet!) reason the URL cannot be created, it will be null, and if the URL is null then you will return null to the getImage() method.

And you are lucky because if you return null to Servoy when he invokes this method, it will simply show the "default" icon and not abandon the idea of using your plugin because of a NPE. Better an effective plugin with a default icon than no plugin at all, don't you think?

You have seen the default icon for a plugin in Servoy, it is this one:



And it's the one used by all the plugins who returned null to the getImage() method, now you know!

So we just completed our first method, and it should look like that:

```
23 public class WhoisPlugin implements IClientPlugin {
24
25     /* (non-Javadoc)
26      * @see com.servoy.j2db.plugins.IClientPlugin#getImage()
27      */
28     public Icon getImage() {
29         URL iconUrl = getClass().getResource("images/whois.png");
30         if (iconUrl != null) {
31             return new ImageIcon(iconUrl);
32         } else {
33             return null;
34         }
35     }
```

**Note** that while we were at it, we got rid of the "// TODO" marker. If you save your effort, you should see that the relevant task disappeared from the "Tasks" view panel. One down, now on to the next!

## 2. getName()

This one is very easy, so I won't spend too much effort explaining it:

```
public String getName() {
    return "whois";
}
```

All you need to do, of course, is to return the name of the plugin as it will appear (and be used for scripting) in Servoy's Solution Explorer plugins node.

## 3. getPreferencePanels()

The next one is the getPreferencePanels() method, and we will leave it to return null since we don't want to give the end user any kind of preference panel in the smart client interface to fiddle with so we have:

```
public PreferencePanel[] getPreferencePanels() {
    return null;
}
```

```
}

```

Note that each time we “implement” a method (even with something as simple as return null – which is refusing to implement it really) we get rid of the // TODO marker so that in the end we should be left with... nothing to do.

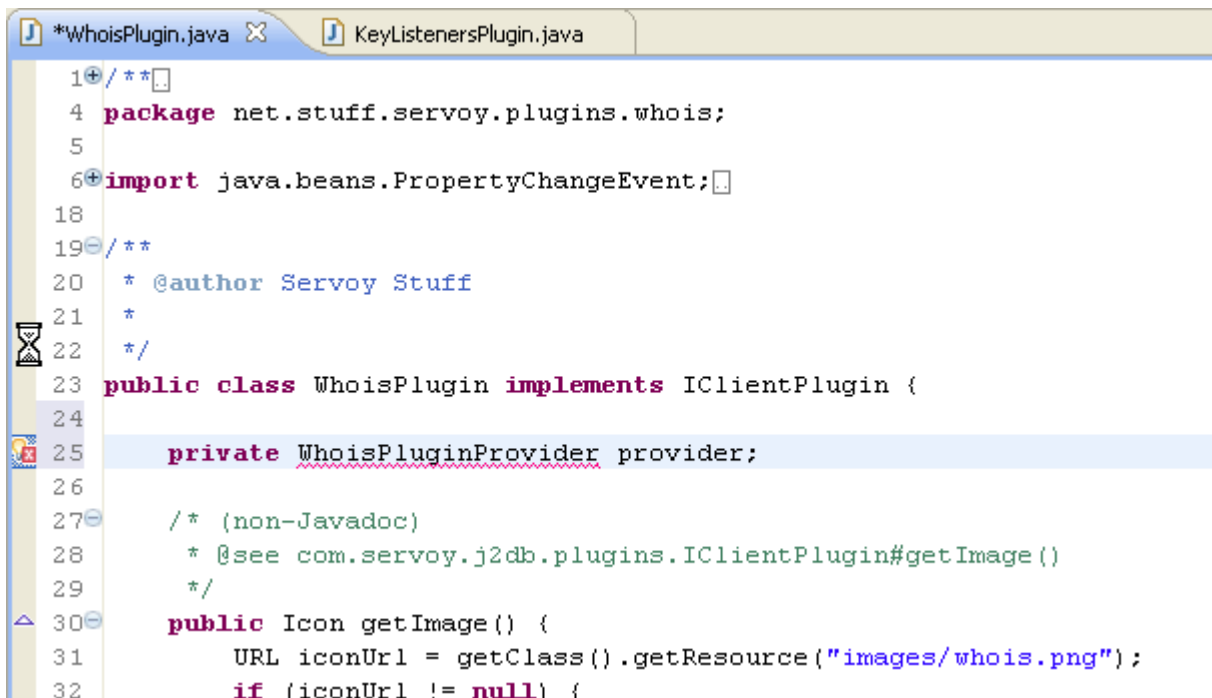
#### 4. *getScriptObject()*

This next one is actually the most interesting one. This is the one which will give your scripts all their power, and to be quite honest we will not code the core of our plugin here, in the IClientPlugin implementing class, but instead we will build a new class based on the IScriptObject interface which is what we are supposed to return here.

We are going to create a class that will implement IScriptObject and that will provide the plugin with scripting (think JavaScript) methods and behaviours, so by convention we will call it a Provider (you are not obliged to do so, but it keeps things neat by stating what each class does), so the name of our new class will be “WhoisPluginProvider”. And instead of creating it the way we already did with the “WhoisPlugin” class, let’s have some fun: we will first declare it in context, then tell Eclipse to create it for us (that’s just another way to do things, and I like you to know a lot of different ways, because depending on what you are doing it might be handier to do it like that).

What am I talking about? Simple, we will use the class before it is even created! Let’s declare the new class as a private object inside the current class by typing:

```
private WhoisPluginProvider provider;
```



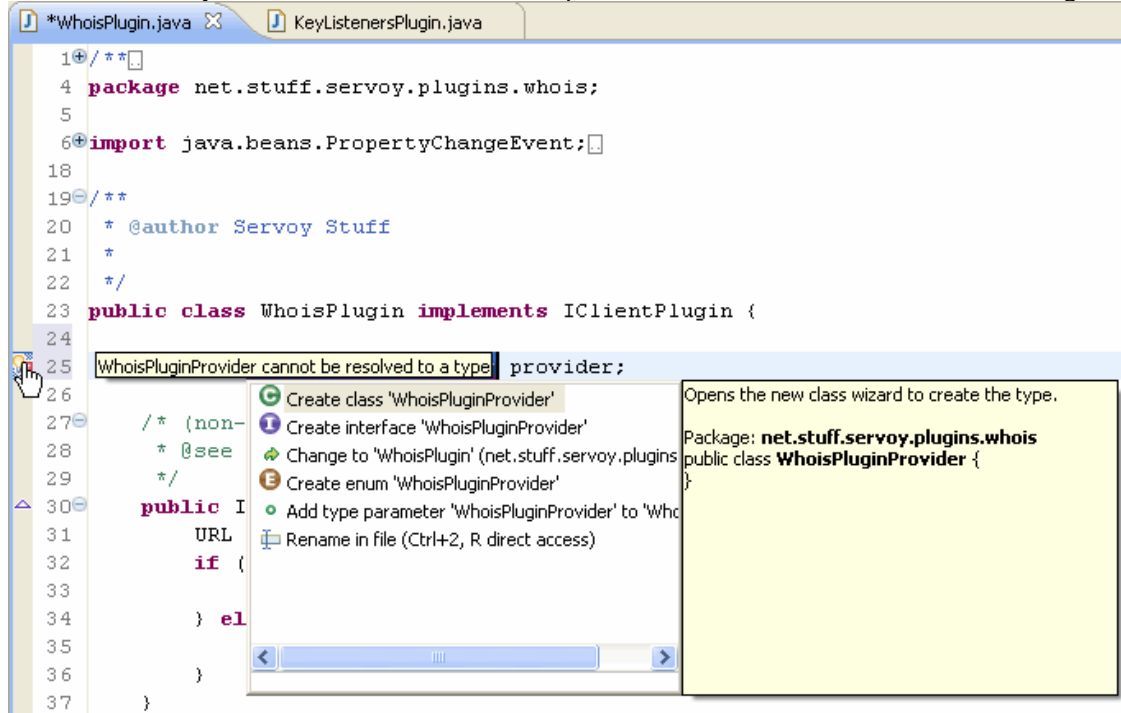
```

1+ /**
4 package net.stuff.servoy.plugins.whois;
5
6+ import java.beans.PropertyChangeEvent;
18
19- /**
20 * @author Servoy Stuff
21 *
22 */
23 public class WhoisPlugin implements IClientPlugin {
24
25     private WhoisPluginProvider provider;
26
27     /* (non-Javadoc)
28      * @see com.servoy.j2db.plugins.IClientPlugin#getImage()
29      */
30     public Icon getImage() {
31         URL iconUrl = getClass().getResource("images/whois.png");
32         if (iconUrl != null) {

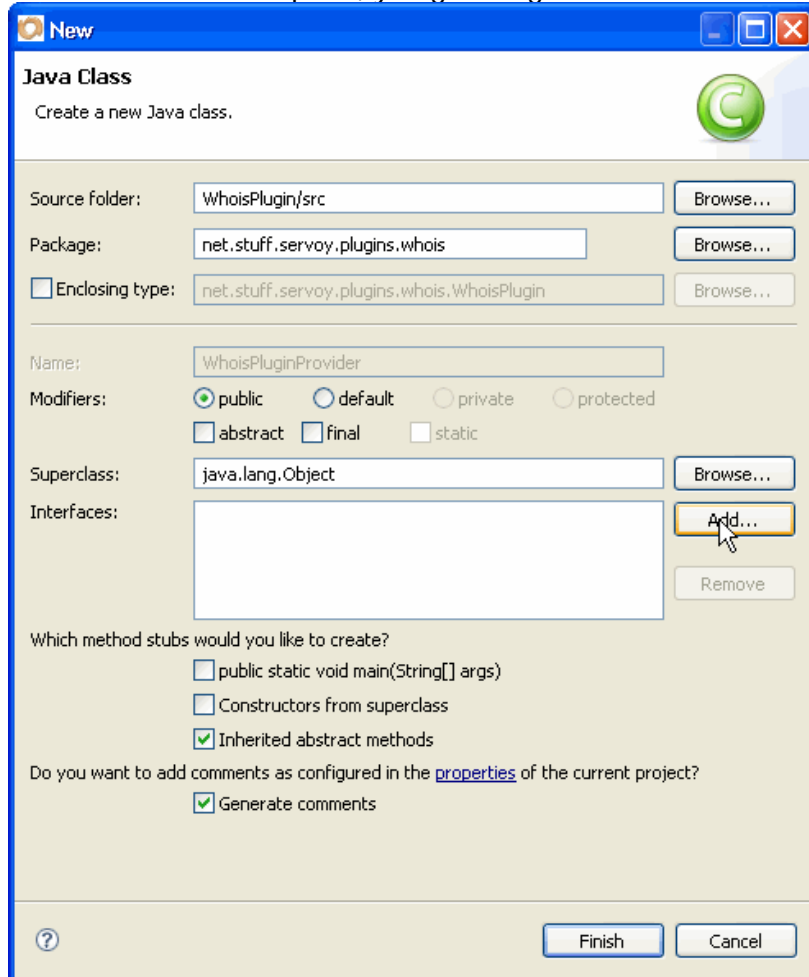
```

Note that the java editor automatically complains about WhoisPluginProvider that can’t be resolved to a type.

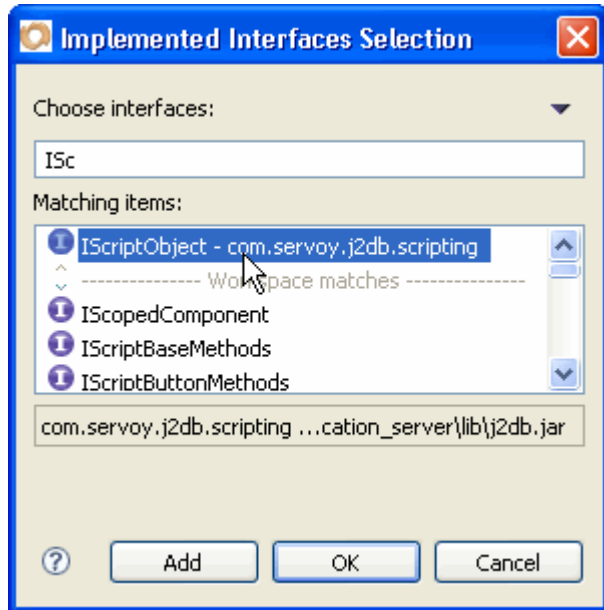
Now use the “quick fix” button, the first option will be to “create class ‘WhoisPluginProvider’”



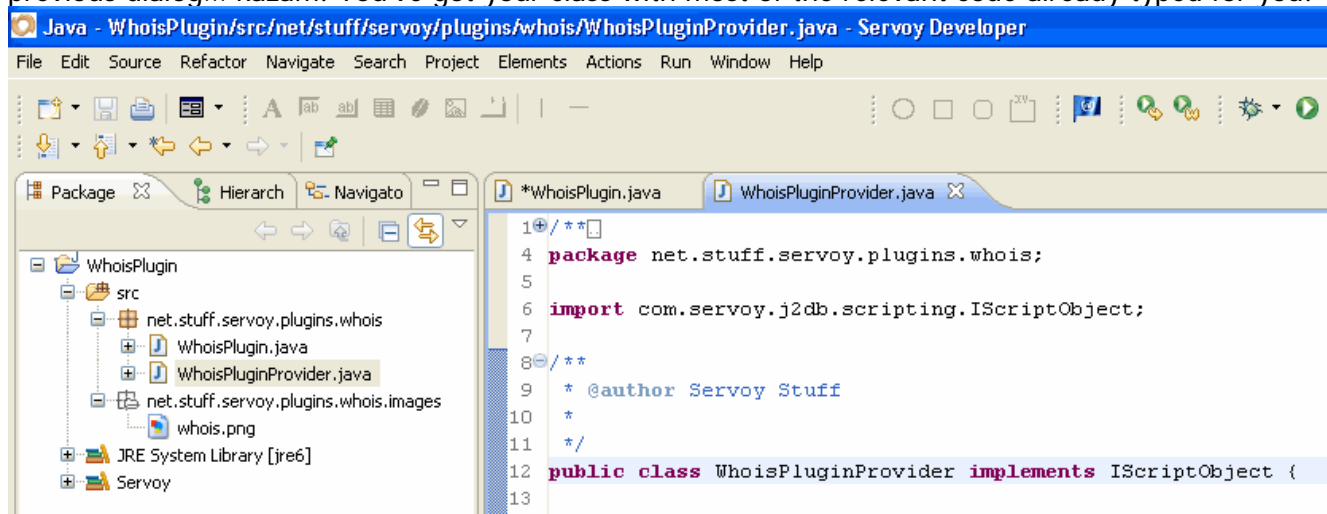
Double-click on that option, you go straight to the “New Class” dialog, and most of it is already filled!



Of course, we know that our WhoisPluginProvider to be truly the object we want to return from our getScriptObject method, will need to implement the IScriptObject interface, so let's add that interface to our class by clicking the "Add..." button and... well you know how to do it now don't you? Type the first letters of the interface name and it will magically appear in the "Matching items" list box:



Double click on the "IScriptObject – com.servoy.j2db.scripting" line, click OK, and then "Finish" on the previous dialog... kazam! You've got your class with most of the relevant code already typed for you!



OK, there are more TODO tasks to implement now, but don't despair, the good news is that with this huge amount of java files (2 all in all!) and this one image file, our plugin has everything it needs to work (except for a few lines of code).

We will create the code of this new WhoisPluginProvider class in the next part of this tutorial, for now let's get back to our WhoisPlugin class where we have left it and finish implementing its methods...

We can now code our `getScriptObject()` method, like this:

```
public IScriptObject getScriptObject() {
    if (provider == null) {
        provider = new WhoisPluginProvider();
    }
    return provider;
}
```

But, you should ask, what is this construct, keeping a reference (a variable - remember how we created the class above by declaring it on top of the file?) to a “provider” variable and testing for null before returning it, instead of simply returning it?

### Two main reasons:

1/ we might want our script object to retain some state. And if we did this:

```
public IScriptObject getScriptObject() {
    return new WhoisPluginProvider();
}
```

We would return a new provider object each time the `getScriptObject()` method is called, so we “keep a reference” to our script object by creating a variable (provider) and keeping it safe and warm inside the plugin object. You can think of the `WhoisPlugin` as the guardian of the `WhoisPluginProvider` object, taking care to return always the same instance of its private object.

2/ we initialize our provider object only if it’s needed.

We could have done this straight away:

```
private WhoisPluginProvider provider = new WhoisPluginProvider();
```

But if the plugin is not used, the object would have been created (and thus would have been eating some resources, CPU/memory) for nothing.

Instead, the first time the caller (the Servoy client) is trying to get hold of the `IScriptObject`, the `WhoisPluginProvider` will be created and not before!

This is a technique called “**Lazy Loading**”, and it is widely adopted to save resources (it eats CPU cycles to create an object as well as it eats memory), the “possibility” of an object is there, but it will only be fulfilled and the object will only be created if and only it is needed (called).

I like Lazy Loading! Being lazy myself, I don’t want to spend too much time trying to solve memory problems for my clients. You should consider adopting this technique too; your clients’ computers client will thank you!

## 5. *initialize()*

We got our `getScriptObject()` done, the next method is `initialize()`.

This one is called by Servoy after the client had start-up, passing in parameters an object of type `IClientPluginAccess` (another interface) which is an open door (not totally opened) to the Servoy client running the plugin.

We will look at that interface in more details and examine what we can do with it in another (larger) tutorial. To simplify here, just note that the only thing it would help us with right now is to know which type of client is running your plugin. But since our script object will be working in any client (it won't have an interface), we don't need that information, so our implementation of this method will be:

```
public void initialize(IClientPluginAccess arg0) throws PluginException {  
    // ignore  
}
```

Yeah, just ignore it, it won't hurt!

## 6. *getProperties()*

Next method: `getProperties()` should return a `Properties` object (we talked about it before, see above). Basically the only required property key/value pair we need to fill for now is the `DISPLAY_NAME` property, so we implement it like this:

```
public Properties getProperties() {  
    Properties props = new Properties();  
    props.put(DISPLAY_NAME, getName());  
    return props;  
}
```

Now look carefully at this code and you will see that this is me being clever ;-)

First we create a `Properties` object (no need for lazy loading here, we don't need to keep a reference to the object so it will be garbage collected when not used by the caller (Servoy) anyway). Then we fill the `props` variable we just created with the key that is provided by the `IClientPlugin` interface (note that by convention constants – static final objects in java – are uppercase). Now the clever (!) thing to note is that instead of filling the `props DISPLAY_NAME` with a string that will be "whois" we call the `getName()` method that we implemented already to return the "whois" String (the name of our plugin).

Why is it better? 3 letters: **DRY**. Don't Repeat Yourself. This is a very simple but effective concept that every developer should try to respect as much as he could, if you already did something somewhere, try to reuse it where it is instead of doing it again somewhere, don't even think of copy/paste. This was much discussed a few years ago with the rise of the Ruby on Rails framework, but they didn't invent the concept, it's as old as good computer engineering.

When the time will come where you'll want to change the name of your plugin, you will only have to do it in one place, and one place only, and it will be changed everywhere it is called.

Now I could also have created a constant like that at the top of the `WhoisPlugin` class:

```
private static final String PLUGIN_NAME = "whois";
```

And then I could have used this constant in the `getName()` method (and the `getProperties` method), except that I didn't feel that I needed to create another variable for that.

It really is a question of taste, constants are good, especially to avoid this “magic numbers” scattered inside your sources (like

```
switch (value) {
    case 11:
        // do something
        break;
    case -2:
        // do another thing
        break;
}
etc...
}
```

These 11 and -2 values might be obvious when you first wrote your code but tend to be totally obscure a few months later, especially if it's someone else that inherits the maintenance of it, but if you used constants, their names will help the poor guy coming after you – if you don't care, remember that the poor guy could be yourself!.

Anyway, you shouldn't abuse of the good things, but always think of the implications in the future of what you are doing today! That's what my grandma always said ☺

### **7. load()**

So we got our `getProperties()` cleverly done, next one is the `load()` method, supposedly called “when the application is starting and before it is started”. Not sure exactly when that is in the lifecycle of the Servoy application(s), anyway we have nothing to initialize especially here, so we can implement it like that:

```
public void load() throws PluginException {
    // ignore
}
```

Told you Java was easy!

### **8. unload()**

This next method is useful to get rid of references to object you might have created, and indeed we created an object, the “provider” variable, holding an object of type `WhoisPluginProvider`, so let's implement the `unload()` method to get rid of it:

```
public void unload() throws PluginException {
    provider = null;
}
```

That's it! Now the garbage collector will be able to dispose of the object. We are environment friendly!

### **9. propertyChange()**

Finally, yes finally! The `propertyChange()` event method, that we inherited from the `IClientPlugin` interface too, it is an event method, meaning that it will be called when a certain event occurs. In this case a `PropertyChangeEvent` will be passed when a “bound” property changes value. I will not explain too much of that, just understand that it is related to the `PreferencePanels` that we could have create for the “Application preferences” dialog in the smart client. Since we didn't create any panel, we don't need to deal with `PropertyChangeEvent` event, so our implementation of this one is:

```
public void propertyChange(PropertyChangeEvent evt) {  
    // ignore  
}
```

That's it, we got our class completed, the IClientPlugin interface is completely implemented, and when we save our java file, it will compile and all the tasks related to the WhoisPlugin class will disappear forever from our "Tasks" view.

To control that you did it everything right, you can download the WhoisPlugin.java file here:  
<http://www.servoy-stuff.net/tutorials/utills/t01/WhoisPlugin.java>

Hope you enjoyed the ride, the next part will be about implementing the IScriptObject interface impersonated by our WhoisPluginProvider class, creating some meaningful JavaScript method to query a whois server, compile, deploy and use in Servoy!

**Patrick Talbot**  
Servoy Stuff  
2009-06-09