Java2Script User's Guide

COLLABORATORS			
	TITLE : Java2Script User's Guide		*REFERENCE* :
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY	Sebastián Gurin	August 8, 2011	

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME
0.00	28 October 2010	Initial revision	sgurin

Contents

List of Figures

Abstract

This document is for Java programmers who want to start programming its web applications in Java language using Java to JavaScript compiler Java2Script .

Chapter 1

About this document

This document is currently a *WORK in PROGRESS*. paragraphs and section contains the work *TODO* mean that the section is to be written or lacks revision.

Document format links (TODO)

The following is a list with summaries about each chapter contents:

- Chapter 2 Introduces Java2Script product
- Chapter 3 Teach the user, step by step how to create its first Java2Script application, how to execute it in the browser, and the basics about Java2Script application projects.
- Chapter 8 deepens on how java code is translated by Java2Script compiler into JavaScript. Details how a Java2Script application is loaded in an html document with javascript.
- In Chapter 4 we give detailed description of all functionalities of Java2Script eclipse plugin, reviewing each component of the plugin and each plugin capability.
- In Chapter 5 we will talk about java language and standard library support in javas2cript. An detailed list of each supported class and method of java.lang, java.util, java.io, etc is presented.
- Chapter 6 explains how to use java libraries supported by Java2Script like SWT, jUnit, etc.
- Chapter 7 explains how to customize Java2Script compiler output using JavaDoc @j2s special tags.
- Chapter 9 gives details about using JavaDoc tag @j2sNative for embedding native javascript code. Gives information and recommendations about using native objects in java code.
- Chapter 10 Describe Java2Script facilities for client-server communication, like RPC, simple pipe, comment and doing normal ajax in Java2Script applications.-
- Chapter 11 contains step by step guide on working with Java2Script plugin sources so the user may debug the plugin and customize it or simply know the plugin's internals.
- Appendix A gives detailed information about Java2Script java language and java runtime javascript emulation, documenting methods of Clazz for java language emulation and ClazzLoader for java runtime emulation.
- Appendix B contains the full text of the license used for releasing this book.

Chapter 2

Introduction

Some actions have an end but no beginning; some begin but do not end. It all depends upon where the observer is standing.

—Frank Herbert - Dune

This document is for Java programmers who want to start programming web applications in Java programming language using Java to JavaScript compiler Java2Script.

We assume that the user is familiarized with the Java programming language.

2.1 What is Java2Script?

TODO: unify and simplfy all content of this section

Java2Script is an Eclipse plugin that helps you write your web applications in Java. This means that your Java codes will be automatically translated into JavaScript which can be executed in a web page. With Java2Script you can transform your java application in a pure javascript application, reusing your java codes in your Rich Internet Applications (RIA), and most of all being able of developing your RIA 100% in java code just like if you were developing another java project.

As you would know, in a common java application this is what happen: First the .java files need to be compiled by the java compiler into .class files. Only then you can execute your program: calling the "java" command the java virtual machine will load and execute the .class files.

As we will see, in a Java2Script application something similar happens. First we need the java2script translator to translate out .java files to .js files and only then we are ready to load and execute the program from javascript in a browser.

A general diagram showing the roles of the Java2Script framework compared with the Java development kit is shown in the followign figure:

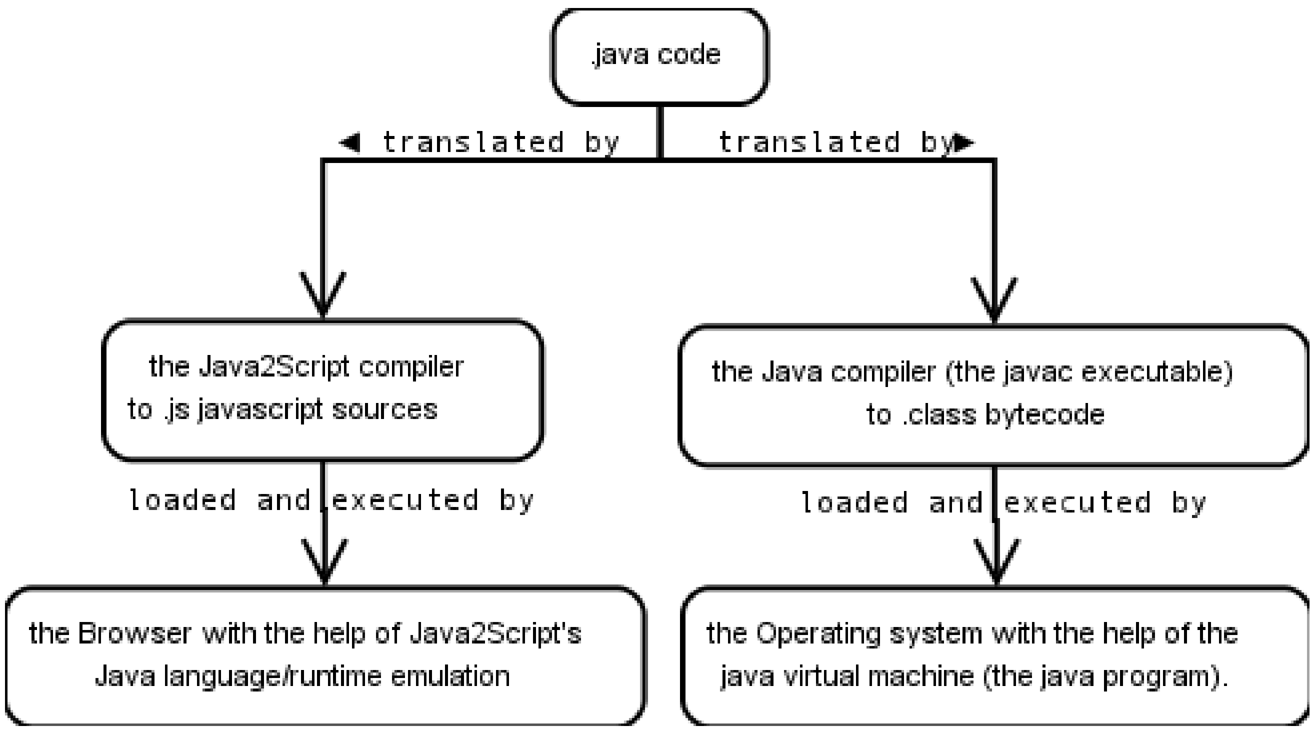

.java code
translated by
translated by
the Java2Script compiler to .js javascript sources
the Java compiler (the javac executable) to .class bytecode
loaded and executed by
loaded and executed by
the Browser with the help of Java2Script's Java language/runtime emulation
the Operating system with the help of the java virtual machine (the java program).

Figure 2.1: Comparing Java2Script and Java JDK roles

For all of this, Java2Script provides with the following components:

- **A java to javascript code translator (compiler)** that translates each .java in a java source folder to a .js file, just like the java compiler javac translate .java files into .class files.
- **Java language and java runtime emulation** Javascript support for loading and executing generated JavaScript in the browser, including java language emulation utilities such as ClassLoader that will lazily load JavaScript code as required, Object Oriented Programming emulation, etc
- **Java core utilities** An implementation of Java SE SDK library is translated, by the Java2Script compiler, to JavaScript. It is available to the Java programmer in HTML document. Currently Apache Harmony Java SE SDK implementation is used.

The following shows a more general architectonic idea involving all the components in developing Java2Script applications.

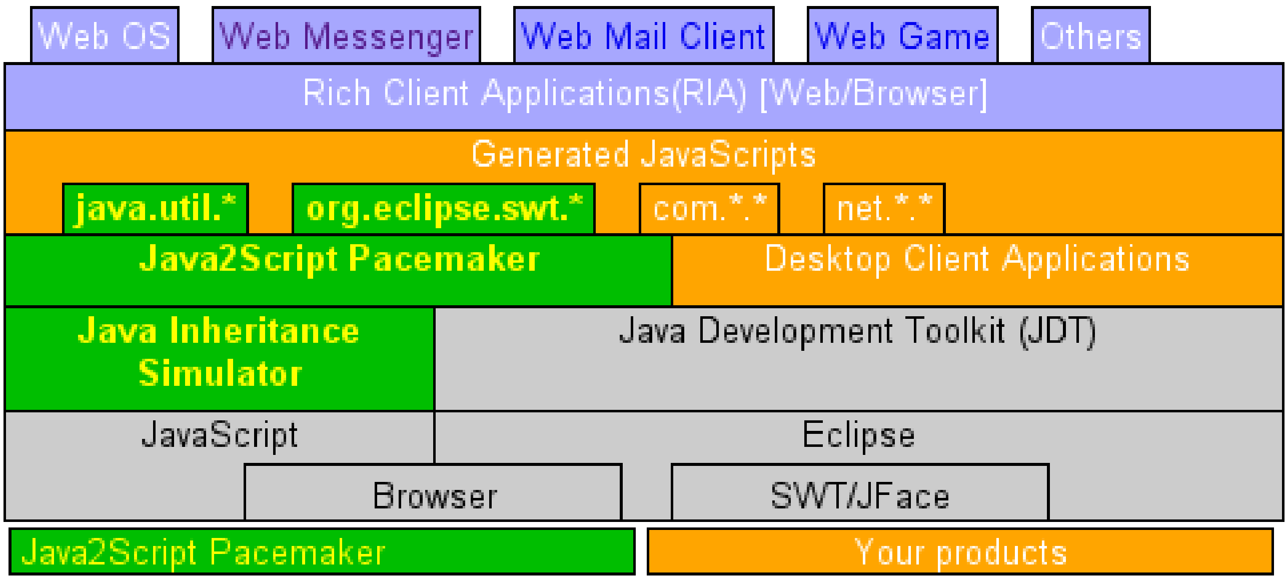


Figure 2.2: Java2Script architecture diagram

For "running" our java programs in a browser, Java2Script also supports a java language emuilation library, letting as easyly configure, load and execute our java translated code inside an html document from javascript.

Also, Java2Script comes with known java libraries like (TODO links) SWT, JUnit and other Java tools translated to JavaScript by Java2Script compiler. All this translated-tools are inside the j2slib folder and are lazily loaded by the J2S class loader when needed by J2S applications. These componenets are:

1. **SWT** SWT version 3 support is available! You can develop your rich internet application without having to learn another GUI toolkit API. More, you can develop applications which can be distributed for both desktop (Java+SWT) and internet!
2. **ajax - simple RPC support.** Aiming to help Java developers balancing and debugging *local* and *remote* procedure calls for AJAX RIAs without knowledge of serialization and deserialization. TODO: http://blog.java2script.org/2006/10/12/java2sc
introduces-simple-rpc/
3. **JUnit** So you do not need to learn or develop another testing framework.
4. **YUI and others** The fact is Java2Script make it easy to port existing Java frameworks to JavaScript and vice versa. With a little more imagination, you can port existing JavaScript frameworks, like YUI, to Java. That is just the mission of the project yui4java that among other things contain tools that easily let the Java programmer to access JavaScript language constructions and Java API for YUI version 2 and 3 and other known JavaScript toolkits.

Note
At this point, it is important to understand that the only mandatory componenets of Java2Script are the first two, the Java to JavaScript compiler and the java language and runtime emulation. All others are optional or replacable by other implementations. For example, as we said, the implemenation for java.lang, java.util java standar library is apache harmony but could be easily replaceable by other implementation because 99% of the java standard library provided by Java2Script is javascript code generated with Java2Script compiler and is for this that we say that the compiler is a principal component of Java2Script.

2.2 Demonstrations

Enough talking, here are some interesting web applications made 100% in Java using Java2Script:

1. **Google Talk Client in Java2Script** Google Talk Client in Java2Script is a pure JavaScript copy of Google Talk Client from non-Google communities. It is first implemented in Java language using Eclipse SWT and Smack (A library for Jabber's XMPP). Then the client is converted to JavaScript with servlet supports. And then it's deployed on a Tomcat server. As Java being platform-independent, and JavaScript is browser-independent, this Google Talk client runs in any modern browsers, such as Firefox, Opera, and Internet Explorer (IE), on Windows, Linux, Mac and others OS platforms. For more details or architectural information, please visit blog "Inside Java2Script".
2. **WeMail Web Mail Client** WeMail Web Mail Client is a free web mail client for Google Mail, Hotmail/Live Mail, Yahoo! Mail, AOL/AIM Mail and other mail providers that support POP/IMAP & SMTP. In case these mail services are blocked by your network firewall in office, you still can access to your mails by using WeMail. WeMail is designed to be used anywhere, Windows/Linux PC, Mac, home or office behind firewall. WeMail uses similar Gmail UI, so those Gmail users should feel much comfortable. WeBuzz.IM team is dedicating to build web mail client services with better security, accessibility and user experience.
3. **SWT Control Examples** SWT Control Examples contain tests for common SWT widgets. You can download the swt-control-examples-1.0.0.zip, unzip it and view the example locally.
4. **Java2Script demo web applications** Java2Script demo web applications contains some of SWT based mentioned demo applications and more, like MSN, Facebook, ICQ, AOL, Yahoo, Jabber, tetris game and other demo applications made with java2script. It shows how several SWT based applications can be presented in one web page.
5. **Advanced Image Viewer** Advanced Image Viewer let you load, view and manipulate your images online. Several image operations are supported, like rotation, zoom, alpha, import/export and advance image filters like blur, emboss, high/low pass filters, edge detection filters and general image convolution based filters are available. This project is a work in progress. For those users who need advanced manipulation technics and filters for online image viewing. 100% client side. Unlike the other java2script projects, it uses YUI instead of SWT for the GUI. And it also uses raphaeljs frameworks, which is ported to Java thanks to yui4java project.

2.3 Comparing Java2Script to other similar technologies

TODO: do not know if this section goes here... TODO write this better.

If you are already familiar with other framework that let you write web pages in java, like GWT [1], you may wonder what are the main differences and similarities of those with Java2Script

When Java2Script is compared to other technologiess like GWT, RAP and other more server side oriented technologies.

```
* it can't be run outside Eclipse,

* it makes a direct compilation from .java to .js file. Concepts like compilation unit,  ↩
    class, classloader are
available

* it has a lazy Java class loader

* it let you write your native
javascript code easily section: the run as dialog: todo. show picture
TODO: explain the porpuse of each item

* use unchecked "Use global j2slib"
only for development. If you want to put your j2slib application in a web
server, you must indicate questions: Â¿is there an official j2slib
published version on some url, so user can use in his/her web
applications without having to have/upload their own j2slib
folder?
```

[1]Google Web Toolkit

```
other comments about j2s vs gwt of mine (not sure of its accuracy):


about gwt vs java2script, I think they are similar tools. IMHO de big desition in this kind ↩
     of projects is ¿how much I have to restrict the java programmer so the resulting  ↩
    javascript code size is acceptable? What I've seen of gwt is that its resulting  ↩
    javascript code size is lower than in java2script. But this is because:

* the translate .java into a very compressed (and ofuscated) javascript code while  ↩
    java2script do an straight and clear translation of .java to .js

* there are restrictions about java.lang and java.util compared to java2script

* only java 1.4 syntax is supported by gwt while in java2script all java syntax versions  ↩
    are supported. - (not so true now!)

* as a result it is easier of porting existing java code and toolkits to javascript with  ↩
    java2script

* java2script provides with metodologies that easily allow to port existing javascript  ↩
    toolkits to java like yui4java

* it is nicely integrated to eclipse IDE: it is very easy to use if you already know about  ↩
    eclipse java projects.

* finnaly somehting totally subjective: I discovered java2script when I was starting to  ↩
    research and develop a product very very similar to java2script (an eclipse plugin based ↩
     on jdt for translating java to javascript). I think at the moment, that was my first  ↩
    motivation of joining java2script team.

Also I've seen other projects for allowing to program RIA in java that heavily use server  ↩
    side for GUI computations (like layouts). Among others :

http://www.eclipse.org/rap/
http://www.fybit.com/products/riatrax4js

This is different to java2script and gwt since the "translation result" is both to  ↩
    javascript+html and serverside.

Also, I've seen other java to javascript compilers that are actually java vm  ↩
    implementations on javascript that "execute" .class files in the browser... a completly  ↩
    different approach to java2script and gwt
```

Chapter 3

Getting Started

"Anticipate the difficult by managing the easy"

—Lao Tzu

In this chapter we will build our firsts web aplications using java2script .... TODO

3.1 A simple Java2Script application

As with any Java application, the first thing is to create an Eclipse project. But instead of creating a common Java application project, we will create a `Java2script application project`. To do so, go to menu `File -> New... -> Other...` and in the dialog select `Java2Script Project`:

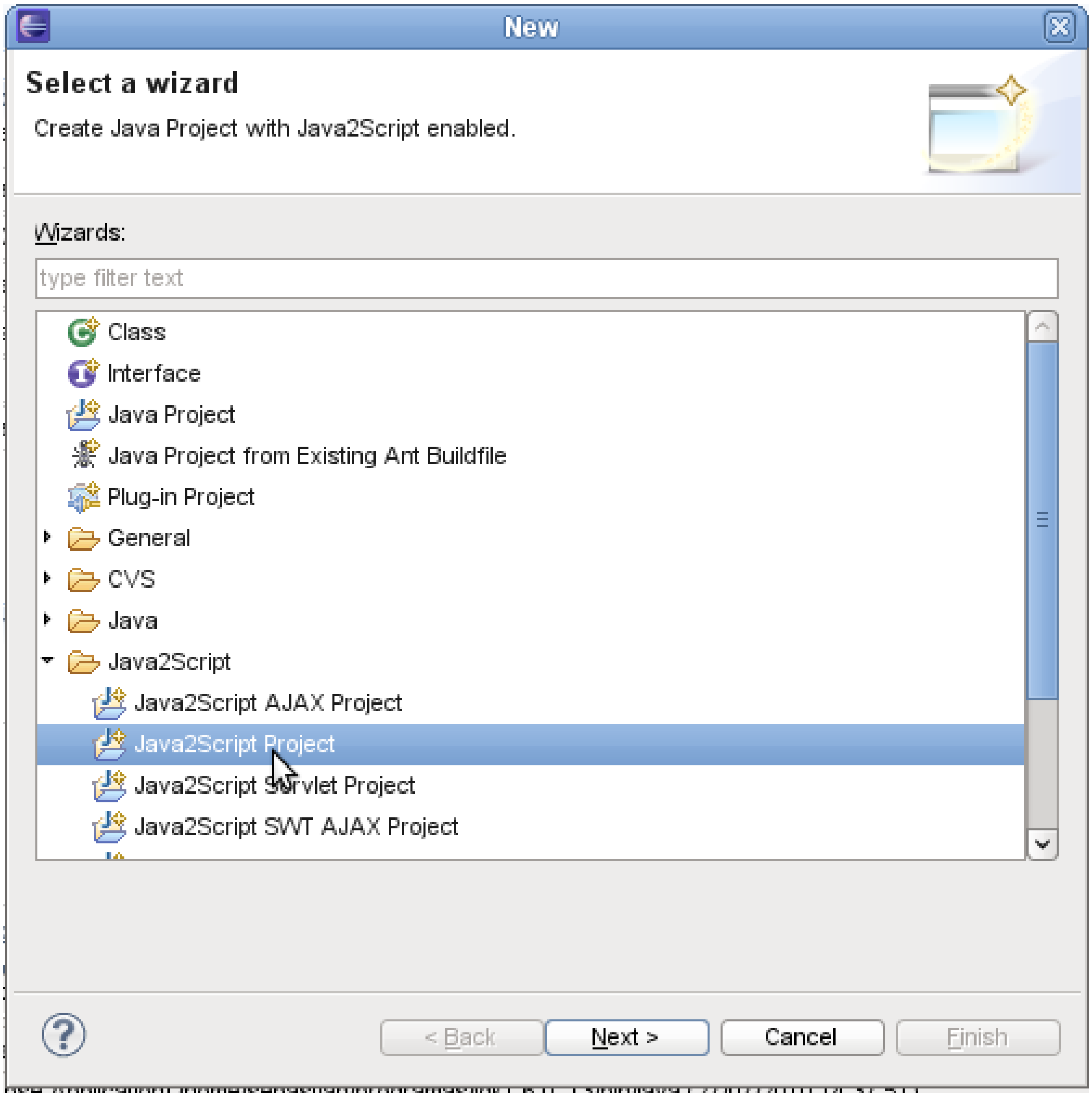


Figure 3.1: Creating a new Java2Script application project

The following steps for finishing creating the Java2Script project are equal to Java project's. Just enter a valid project name and click next/finnish. Congratulations you have created your first Java2Script project!

Note

Java2Script projects are just like common Java projects, but with the Java to JavaScript compiler enabled. This means that your .java files will also be compiled to JavaScript. So the application can be executed inside an HTML document as we will see below. By the way, the generated JavaScript codes will be at the same location as generated .class files, by default at `YOU_PROJECT/bin` folder.

Now, we will create a simple Java hello world program and see how easily can be executed in an HTML document. First create a Java package like `my.first.project` and add a Java class named `HelloWorld` inside it with the following code:

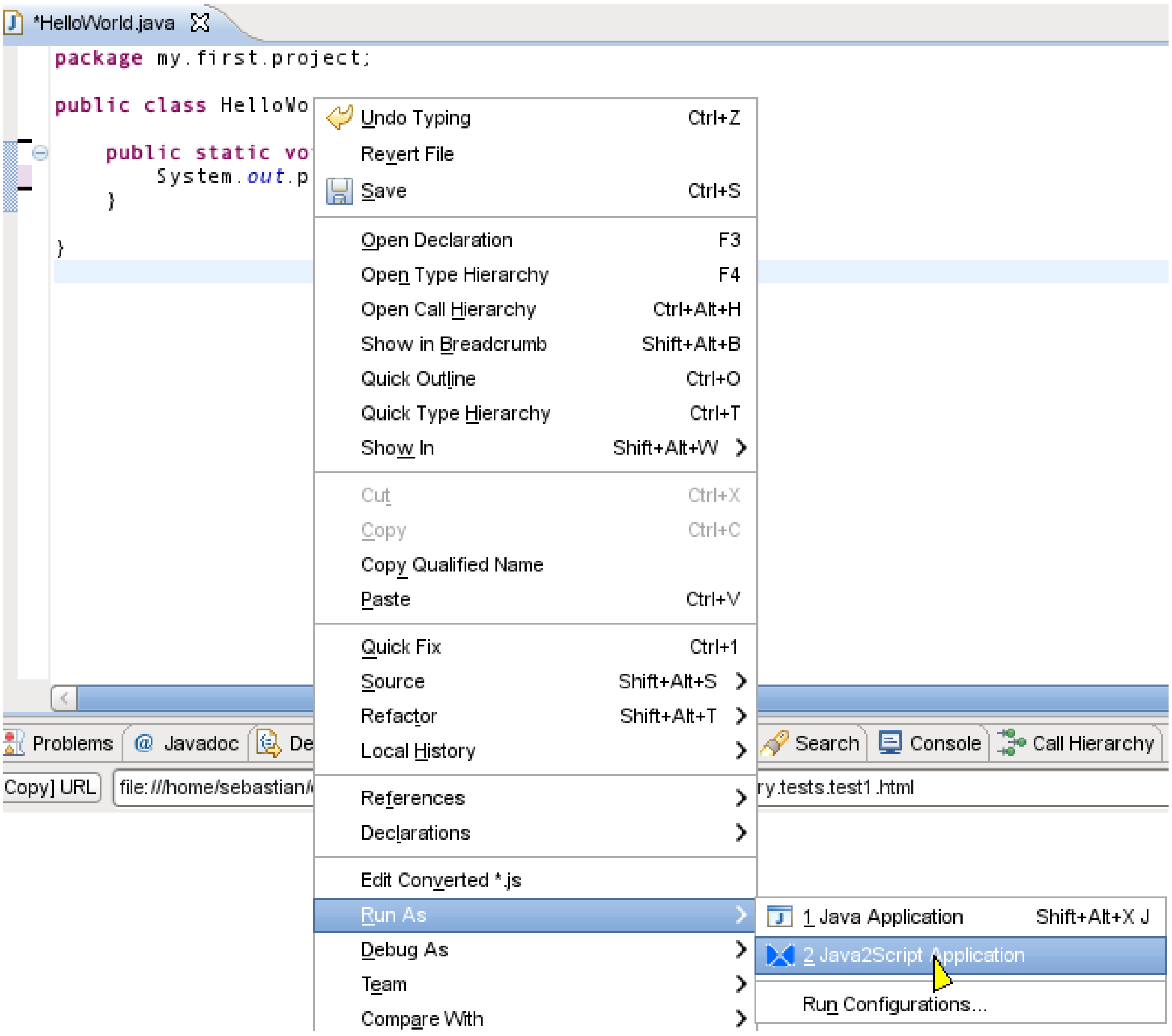
```
package my.first.project;

public class HelloWorld {

  public static void main(String[] args) {
    System.out.println("hello world");
  }

}
```

Save the file with `CTRL + S`. Now we will execute the `HelloWorld` class as a Java2Script application. An HTML document named `my.first.project.HelloWorld.html` will be created that launch our little Java application. You can launch your Java2Script application in the same way you launch a normal Java application: right click you Java code `-> Run As -> Java2Script Application`:

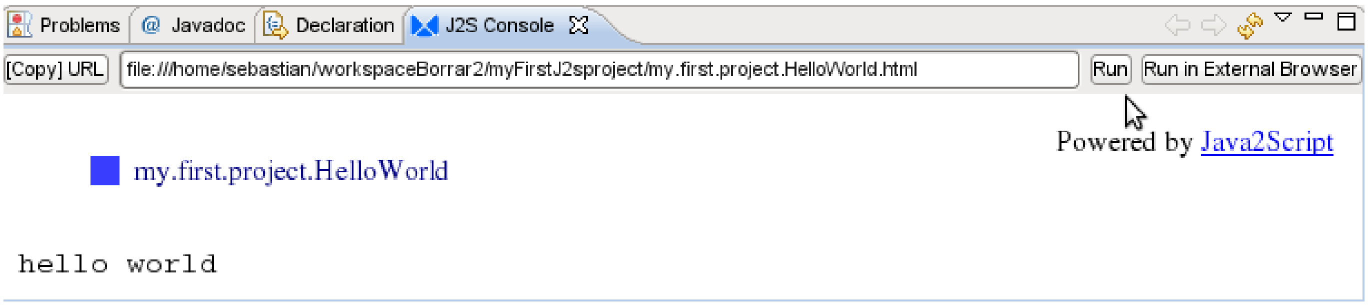


```
package my.first.project;

public class HelloWo
    public static vo
        System.out.p
    }
}
```

Figure 3.2: Launch a Java2Script application

When you run a Java class as a Java2Script Application, the J2S console Eclipse view will be opened and it will show an HTML document which will execute your Java application. This view is a simple browser and in fact you can also open the generated HTML document with your favourite browser:


Problems
Javadoc
Declaration
J2S Console
[Copy] URL
file:///home/sebastian/workspaceBorrar2/myFirstJ2sproject/my.first.project.HelloWorld.html
Run
Run in External Browser
Powered by Java2Script
my.first.project.HelloWorld

```
hello world
```

Figure 3.3: Launch a Java2Script application - the J2S Console view

Congratulations, you have just executed your first Java2Script application! As you can see, java2script simulate the `System.-out.println` method call pending a paragraph with text to the html document. Another thing to notice, is that if you refresh your project, you will see a new html file at the root

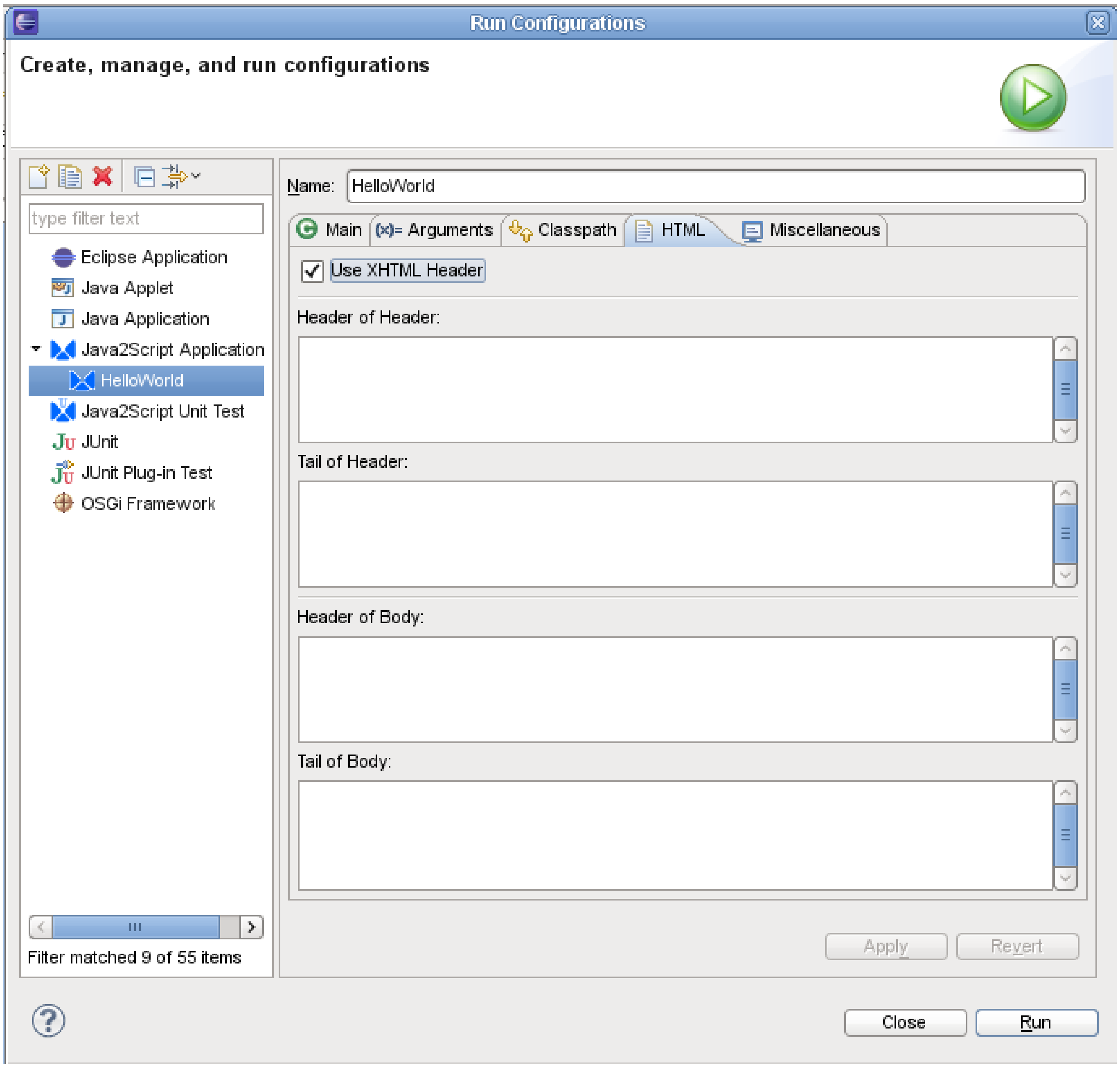
Tip
Remember, if you just make a change in a .java file and save the file, the J2S compiler will regenerate the javascript code and you only need to refresh the HTML document in the browser. If you only change a .java file, you don't need to run as.. the application again, only refresh the document in the browser. Try it, copy the url of the html document from the J2S console, and paste it in your stem browser. Then change the System.out string in HelloWorld.java file and save it. Now refresh the browser and you should see the change reflected.

3.2 Configuring your Java2Script application launcher.

A very important part of Java2Script plugin is the application launcher configuration. By default, Java2Script applications will be launched from local filesystem and will point to the `j2slib.js` file that is inside your eclipse's `plugins/` directory.

When you run your application the first time, a Java2Script application launcher configuration is created. For editing it, you just rught click on you java file -> Run Configgurations... As you may see, there are two new tabs in the application launcher configuration.

The first tab, `HTML`, is used to introduce html code at the `<head>` or `<body>` html elements of the generated html document, like scripts, CSS stylesheets, etc. We will call this tab the 'HTML configuration tab':



Run Configurations

Create, manage, and run configurations

type filter text

- Eclipse Application
- Java Applet
- Java Application
- Java2Script Application
  - HelloWorld
- Java2Script Unit Test
- JUnit
- JUnit Plug-in Test
- OSGi Framework

Filter matched 9 of 55 items

Name: HelloWorld

Main | (x)= Arguments | Classpath | HTML | Miscellaneous

[✓] Use XHTML Header

Header of Header:

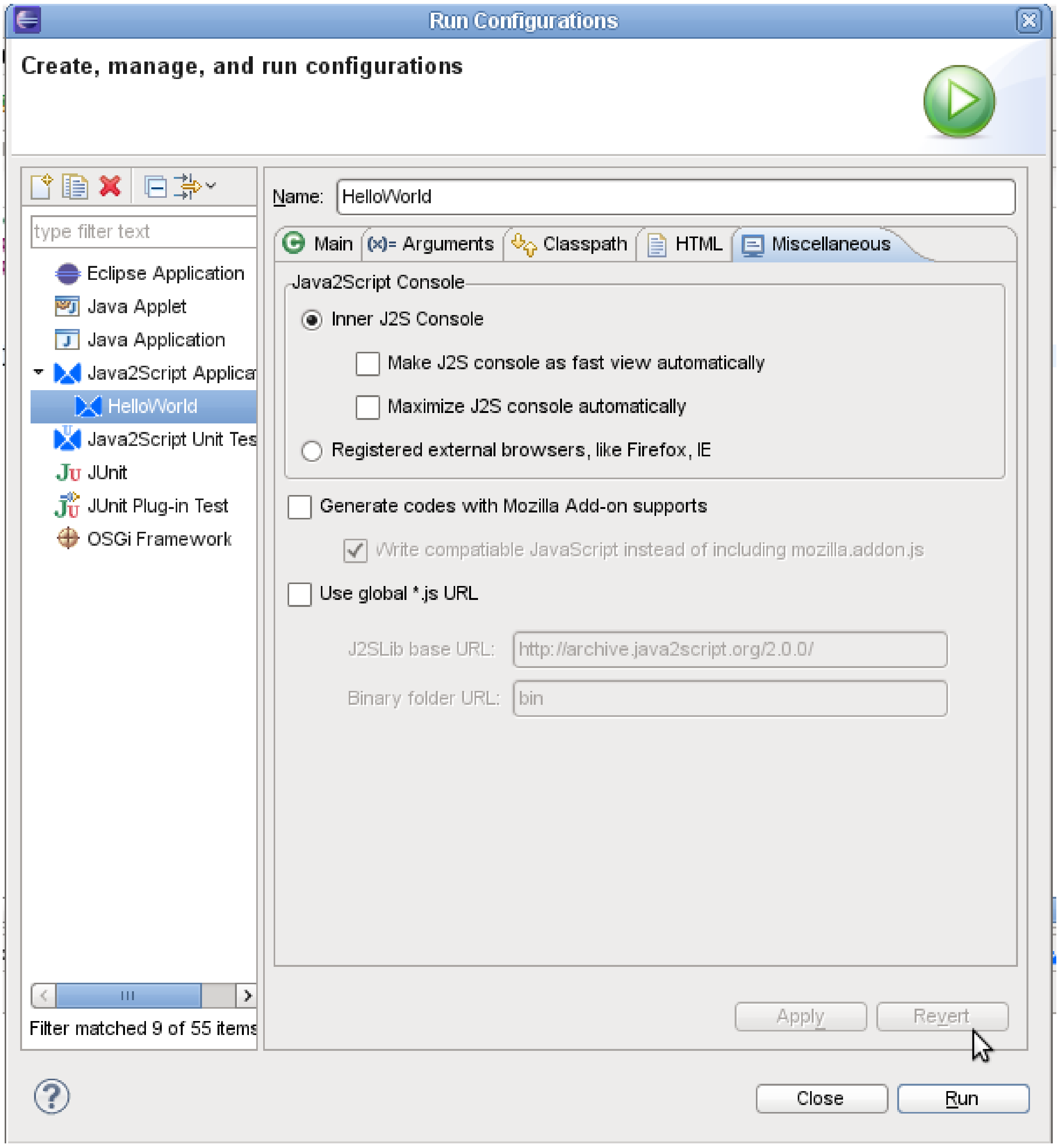
Tail of Header:

Header of Body:

Tail of Body:

Figure 3.4: Launch a Java2Script application - the J2S Console view

The `Miscellaneous` tab is used to specify, among other things, the location of the `j2slib.js` file, the location of generated `.js` files (the `bin/` folder URL), firefox addon support, etc:



Run Configurations

Create, manage, and run configurations

type filter text

- Eclipse Application
- Java Applet
- Java Application
- Java2Script Applica
  - HelloWorld
- Java2Script Unit Tes
- JUnit
- JUnit Plug-in Test
- OSGi Framework

Filter matched 9 of 55 items

Name: HelloWorld

Main | (x)= Arguments | Classpath | HTML | Miscellaneous

Java2Script Console

- (●) Inner J2S Console
  - [ ] Make J2S console as fast view automatically
  - [ ] Maximize J2S console automatically
- ( ) Registered external browsers, like Firefox, IE

[ ] Generate codes with Mozilla Add-on supports

[✓] Write compatiable JavaScript instead of including mozilla.addon.js

[ ] Use global *.js URL

J2SLib base URL: http://archive.java2script.org/2.0.0/

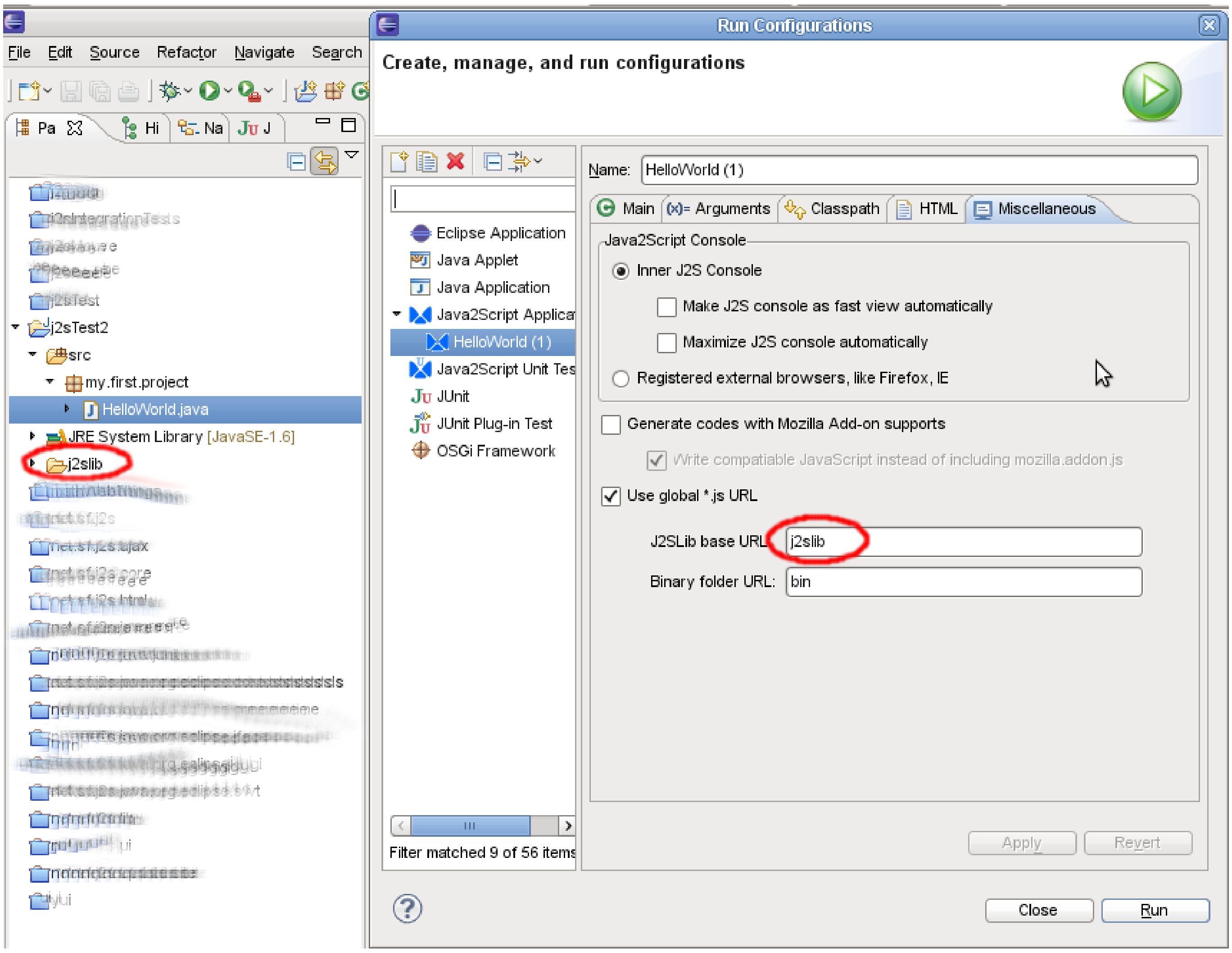
Binary folder URL: bin

Figure 3.5: Launch a Java2Script application - the J2S Console view

The most important item to configure here is the location of `j2slib.js` file. By default, Java2Script are launched pointing to `plugins/` internal `j2slib.js` filesystem version, for example :

```
<script type="text/javascript" src="../../j2s_3.6_workspace/net.sf.j2s.lib/j2slib/j2slib ↩
    .z.js">
```

This is aceptable for developing because the Java2Script application will be loaded fast from filesystem. Nevertheless, you should change this if you want to deploy your application to a web server. A common practice is to copy your `eclipseclassichelios/plugins/net.sf.j2s.lib_2.0.0/j2slib/` folder inside your Java2Script project and configure the application launcher to use a Global .js URL for your j2slib base:



Create, manage, and run configurations

Name: HelloWorld (1)

Java2Script Console
- [✓] Inner J2S Console
  - [ ] Make J2S console as fast view automatically
  - [ ] Maximize J2S console automatically
- [ ] Registered external browsers, like Firefox, IE

[ ] Generate codes with Mozilla Add-on supports
[✓] Write compatiable JavaScript instead of including mozilla.addon.js
[✓] Use global *.js URL

J2SLib base URL: j2slib
Binary folder URL: bin

Figure 3.6: Launch a Java2Script application - the J2S Console view

Chapter 4

The J2S plugin

Everything that can be counted does not necessarily count; everything that counts cannot necessarily be counted.
—Albert Einstein

TODO: about the plugin:

```
8. Support creating Java2Script projects by wizard, including
Servlet project
9. Support Eclipse versions from 3.3 to 3.6
```

4.1 Java2Script project

this section describe project configuration, project properties page, .j2s file, in .j2s file

TODO

4.2 J2S Java to JavaScript compiler

The most important component is the java to javascript code translator. This is an eclipse plugin, based on eclipse java develpment tools (JDT / http://www.eclipse.org/jdt/), that will compile your eclipse java project to javascript.

```
1. Object oriented JavaScript simulator is introduced
2. JavaScript ClassLoader is built inside
3. Eclipse JDT based Java to JavaScript compiler, support Java
1.4 and Java 5 features, including generic, enums, static imports,
enhanced loops
4. Support incremental building for Java projects
5. Support Hotspot JavaScript swapping for debugging
6. Support most classes in java.lang.*, java.utils.* and others
packages
7. Support native JavaScript through @j2s* Javadoc in Java sources and java anotations
```

TODO: do this better. Ideas: * talk about compiler supported java versions, * in section chap-java-to-js-translation will talk about code tranlation *

4.2.1 Compiler configuration

TODO: how the compiler can be configure with the .j2s project's file

4.3 Application launching

this section describe j2s app launcing in detail

TODO

4.3.1 Using templates

this section describe j2s application renderization throw templates, in this case doucments the velocity contribution.

TODO

4.4 Debugging

how turn on debuggin?

what happens when debug is on?

4.4.1 JavaScript ClassLoader and Hotspot

If you are familiar with Java debugging, you must know there is a technology called "Hotspot". One feature of Hotspot is to replace old classes bytecodes with a new ones that are generated by dynamic compilers. This feature helps developers a lot in debugging. That is to say, when a developer load a very complex application in debugging mode, he want to modify the sources a little, he can just do it, the compiler will compile those related classes and notify classloader to load those affected classes bytecodes. So the changes are loaded in an application being executed without the need of reloading it. It saves lots of time by avoiding closing, reopening and waiting big applications again and again. This is very convenient when comparing to those static compiled applications written by C or C++ languages.

Let's try it with an example. The following java test, creates an html button with a click event handle. Each time the user clicks the button, the instance method `doCLickAction()` is called.

```
package org.sgx.j2s.js;

import org.sgx.j2s.html.HTMLUtils;

public class HotSpotTest1 {
  int counter = 0;

  public static void main(String[] args) {
    new HotSpotTest1().test();
  }

  public void test() {

    //creates an html button element in document.body, with label "click me" and
    //with an event handler that will call doCLickAction() instance method

    HTMLUtils.createButton(JsUtils.document().body, "click me", new Runnable() {
      public void run() {
        counter++;
        doClickAction();
      }
    });
  }

  protected void doClickAction() {
    System.out.println(counter+" say hello!");
  }
}
```

Now we will use J2S Hot Spot for debugging the application making changes to sources that will be reflected in the html document without having to reload the application. For debugging a J2S application with HotSpot, we use Debug As... instead of Run As... in the context menu:

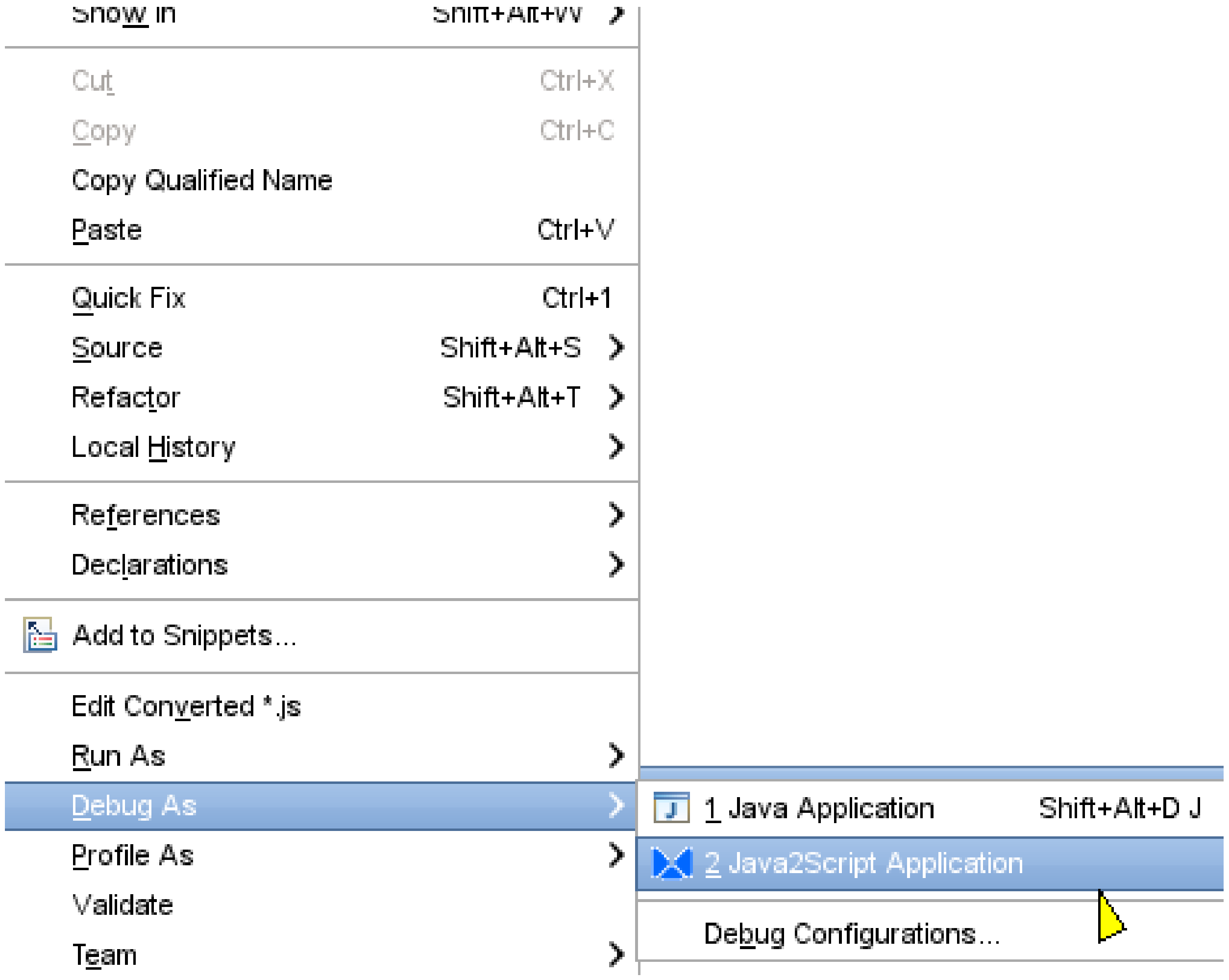
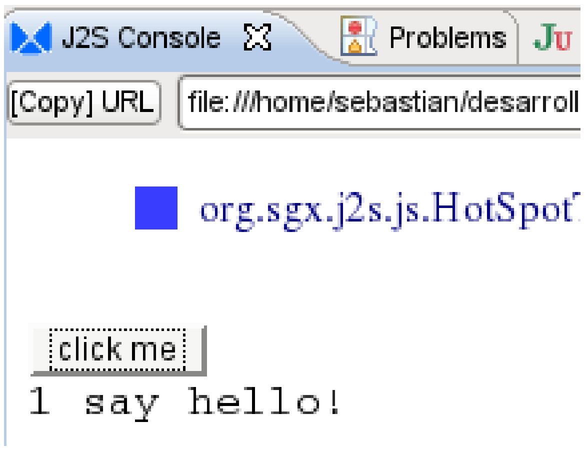


Figure 4.1: Debug As... Java2Script application context menu

This will execute our application that will show a button labeled "click me". If we click the button once a message is printed:



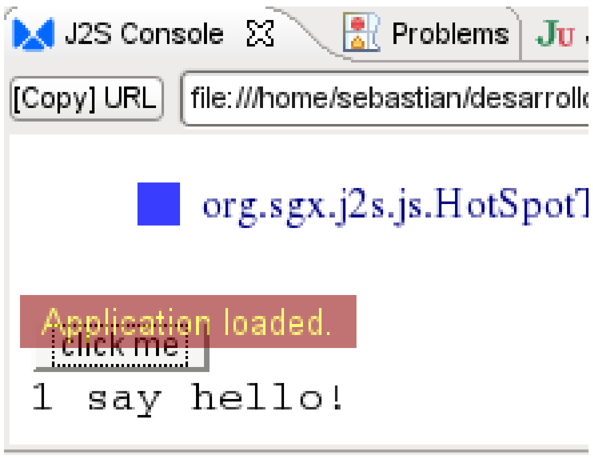
```
1 say hello!
```

Figure 4.2: HotSpot debugging the application - first screenshot

Now the interesting part. Change the message printed in the method `doClickAction()`, for example:

```
protected void doClickAction() {
  System.out.println(counter+" say hello CRUEL WORLD!");
}
```

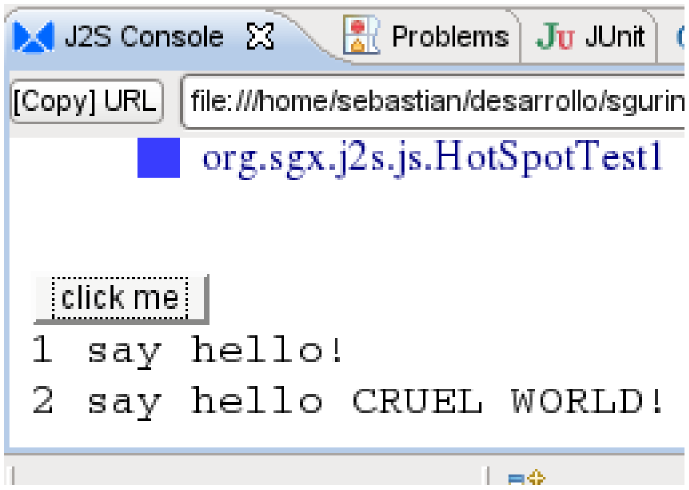
Save the changes and keep looking at your application at the J2S console. After a few seconds, you should see a red label "Application loaded":



```
1 say hello!
```

Figure 4.3: HotSpot debugging the application - application change notification

That means the J2S HotSpot mechanish has detected changes in classes of current application, and the application has loaded the new changes. So, we can click the button again, and will see the new message printed:


J2S Console
Problems
JUnit
[Copy] URL
file:///home/sebastian/desarrollo/sgurin
org.sgx.j2s.js.HotSpotTest1
click me

```
1 say hello!
2 say hello CRUEL WORLD!
```

Figure 4.4: HotSpot debugging the application - changes were applied

W just were able of modify the application and reflect those changes all without restarting our application, or in other words, we have just make a "hot change" in our application.

Note
Just like in java, only changes to instance variable and instance methods will be supported by HotSpot. Changes to static methods, or changes in class signatures (like adding/removing/renaming) methods are not supported by HotSpot.

How is this accomplished?

Java2Script also implements JavaScript's Hotspot technology. It is not complicate, because JavaScript is already a very robust and convenient language to do so. All Java2Script implementation is to clean those classes' declaration inside JavaScript class inheritance system simulator. And then reload the *.js using Java2Script's classloader. In the implementation, classloader does not change class or object's prototype when a reloaded class is redefined, it's possible to keep all classes relationships without breaking those already instantiated instances.

To trigger Java2Script class simulator to Hotspot swapping, there is a thread at the plugin side, trying to load an updating JavaScript classes list from Java2Script compiler in Eclipse. Its work is simple, just trying to load http://127.0.0.1:1725/<session>.js. And the server listening on default port 1725 is started by Java2Script compiler, called "Java2Script Inner Hotspot Server (J2SIHS)". When a compiling occurs, compiler will notify this server that a class is updated, and the server will add the information to the list. Once a JavaScript request arrives, it will send out the list according to the request session id. And when the JavaScript client thread gets the updated classes list, it will try to unload related classes and reload them. That is the rough procedure of Java2Script Hotspot technology.

By using Hotspot, I think it is much more convenient for me to develop JavaScript RIA in Java codes than before, especially in developing SWT applications.

4.5 Libraries in Java2Script

In this section we will detail mechanisms provided by Java2Script plugin for using and distributing java software. How java software components can be distributed so other Java2Script users can use them in their J2S porojects.

Suppose you develop a very useful software component in Java2Script and now you want to distribute so other can use it in their own java2script projects. In normal java project we normally put our .class files inside a .jar (Java ARchive) file, and distribute the .jar. That is enought, but not in a java2script project where, besides .class files you must also can distribute generated javascript files.

Of course you can always distribute your library sources, and let the users import the osources into their own projects, perhaps in differents sourcefolders, but that is generally a good idea when you want to encapsulate your library api and implementation.

Fortunately, Java2Script provide with flexible mechanism of library definition. Also as we will see, Java2Script support for defining how the library must loaded

TODO: talk about j2slib and how swt, junit etc are inside. TODO: talk about .j2x files and package.js files that le definehow alibrary must be loaded. Also, try and if works to dev a library, put it in other folder that j2slib, with a pacakge.js and lib.j2z file, and show how oackage.js is evaluated, and how to use it for loading one, concatenating several files, etcx

TODO: p/d: future extension packer can be mention here for intelligent generattion of big .js file .

4.6 Command line API

TODO: document here the expiremental support for j2s console api. ?

Chapter 5

Java Support

The best way to predict the future is to implement it.

—Alan Kay

Of course, there are java classes that cannot be emulated in the browser. For example, the class java.util.File: there is no way in a browser to read and write a file in the filesystem, so the classes related to filesystem like java.lang.File, java.lang.FileOutputStream are not supported by Java2Script. The same way, it is not possible to use multiple threads or processes in a javascript program and so classes related to threads like java.lang.Thread are not supported.

5.1 Using unsupported java classes

Let's see what happen when we try to use unsupported java classes in our Java2Script applications. Let's create java class with the following content:

```
package foo.bar;

import java.io.File;

public class WontWork {
  public static void main(String[] args) {
    new File("foo.txt");
  }
}
```

Let's see if Java2Script compiled the class. Save the file, right click and choose "Edit converted *.js" option from the context menu:

```
&#xfeff;Clazz.declarePackage ("foo.bar");
Clazz.load (null, "foo.bar.WontWork", ["java.io.File"], function () {
c$ = Clazz.declareType (foo.bar, "WontWork");
  c$.main = Clazz.defineMethod (c$, "main", function (args) {
    new java.io.File ("foo.txt");
  }, "~A");
});
```

So, as we can see in line 5 of the generated js file, the java class is translated anyway, no matter if it contains or reference a not supported java class. The compiler won't fail, but if we execute the class as a J2S application, the generated html will contain a message error like the following:

```
[Java2Script] Error in loading ../path/to/j2slib/java/io/File.js!
```

More, if we open the html document with firefox, the following javascript error will be thrown in the Firefox error console:

```
Error: java.io.File is not a constructor
Source File: file:///path/to/project/bin/foo/bar/WontWork.js
Line: 6
```

Remember we have said that one of the main porpuses of J2S is to let the user to reuse Java codes inside web applications. So, this feature of not failing at compile time but at run time can be usefull when trying to compile an existing Java library or framework to javascript because the Java code will be compiled no matter if it contains not supported java classes. Also, generated javascript will run OK as long as the unsupported classes are not needed. But will fail when trying to load an unsupported class with a message like we have showed.

Important
Java classes referencing unsupported java classes will not fail to compile. What will fail is loading unsuported classes when the J2S application is launched in an HTML document.

5.2 JRE Emulation Reference

TODO. instead only listing files, argument how well is supported each class with .,..

Java2Script includes a library that emulates a subset of the Java runtime library. The list below shows the set of JRE packages, types and methods that Java2Script can translate automatically. Note that in some cases, only a subset of methods is supported for a given type.

- Package java.lang
- Package java.lang.annotation
- Package java.lang.reflect
- Package java.io
- Package java.util

5.3 Package java.lang

Almost all java language classes are supported. The main restriction is with classes related with Threads, like java.lang.Thread,

```
workspace/net.sf.j2s.java.core/src/java/lang> ls
AbstractMethodError.java            CloneNotSupportedException.java     ←
    InstantiationException.java      SecurityException.java
AbstractStringBuilder.java          Comparable.java                    Integer.js  ←
                          Short.js
annotation                          Console.js                         InternalError.java  ←
                  StackOverflowError.java
Appendable.java                     Copy of Boolean.js                  ←
    InterruptedException.java        StackTraceElement.java
ArithmeticException.java            Double.js                          Iterable.java  ←
                       StrictMath.java
ArrayIndexOutOfBoundsException.java  Encoding.js                       LinkageError.java  ←
                   StringBuffer.java
ArrayStoreException.java            Enum.java                          Long.js  ←
                             StringBuilder.java
AssertionError.java                 Enum.js                             ←
    NegativeArraySizeException.java  StringIndexOutOfBoundsException.java
```

```
Boolean.js                          Error.java                          ←↩
    NoClassDefFoundError.java       String.js
Byte.js                             ExceptionInInitializerError.java   NoSuchFieldError. ←↩
    java            ThreadDeath.java
Character.java                      Exception.java                      ←↩
    NoSuchFieldException.java       ThreadGroup.java
CharSequence.java                   Float.js                           NoSuchMethodError. ←↩
    java           Thread.java
ClassCastException.java             IllegalAccessError.java             ←↩
    NoSuchMethodException.java      Throwable.java
ClassCircularityError.java          IllegalAccessException.java         ←↩
    NullPointerException.java       TypeNotPresentException.java
ClassExt.js                         IllegalArgumentException.java       ←↩
    NumberFormatException.java      UnknownError.java
ClassFormatError.java               IllegalMonitorStateException.java  Number.js  ←↩
                          UnsatisfiedLinkError.java
Class.js                            IllegalStateException.java         OutOfMemoryError. ←↩
    java            UnsupportedClassVersionError.java
ClassLoader.js                      IllegalThreadStateException.java   Readable.java  ←↩
                      UnsupportedOperationException.java
ClassLoaderProgressMonitor.js       IncompatibleClassChangeError.java  reflect  ←↩
                            VerifyError.java
ClassNotFoundException.java         IndexOutOfBoundsException.java     Runnable.java  ←↩
                      VirtualMachineError.java
Cloneable.java                      InstantiationError.java            RuntimeException. ←↩
    java
```

5.3.1 Package java.lang.annotation

TODO

5.3.2 Package java.lang.reflect

There is limited support for java reflection. TODO: what is not supported?

5.4 Package java.io

This is the package with more restrictions. Nevertheless, there is support for some kind of Streams that can be emulated in javascript.

```
/sgurin_workspace/net.sf.j2s.java.core/src/java/io> ls
BufferedInputStream.java    CharConversionException.java  FilterOutputStream.java       ←↩
    NotSerializableException.java  StringBufferInputStream.java
BufferedOutputStream.java   Closeable.java                Flushable.java                ←↩
    ObjectStreamException.java     StringReader.java
BufferedReader.java         DataInput.java                InputStream.java              ←↩
    ObjectStreamField.java         StringWriter.java
BufferedWriter.java         DataOutput.java               InterruptedIOException.java   ←↩
    OptionalDataException.java     SyncFailedException.java
ByteArrayInputStream.java   EOFException.java             InvalidClassException.java    ←↩
    OutputStream.java              UnsupportedEncodingException.java
ByteArrayOutputStream.java  Externalizable.java           InvalidObjectException.java   ←↩
    Reader.java                    UTFDataFormatException.java
CharArrayReader.java        FileNotFoundException.java    IOException.java              ←↩
    Serializable.java              WriteAbortedException.java
CharArrayWriter.java        FilterInputStream.java        NotActiveException.java       ←↩
    StreamCorruptedException.java  Writer.java
```

5.5 Package java.util

In general, all kind of collections and utils are supported. Some utilities have limited support, like Date and Regexp. There is no support for DateFormat stuf. Utilities related to unsuported features like filesystem and Threads, like Timer, TimerTask are not supported. TODO: folrmalize this.

```
/sgurin_workspace/net.sf.j2s.java.core/src/java/util> ls
AbstractCollection.java                      HashSet.java                             ↩
    MissingResourceException.java
AbstractList.java                            Hashtable.java                           ↩
    NoSuchElementException.java
AbstractMap.java                             IdentityHashMap.java                     ↩
    Observable.java
AbstractQueue.java                           IllegalFormatCodePointException.java     ↩
    Observer.java
AbstractSequentialList.java                  IllegalFormatConversionException.java    ↩
    Properties.java
AbstractSet.java                             IllegalFormatException.java             Queue. ↩
    java
ArrayList.java                               IllegalFormatFlagsException.java         ↩
    RandomAccess.java
Arrays.java                                  IllegalFormatPrecisionException.java    Random. ↩
    java
Collection.java                              IllegalFormatWidthException.java        regex
Collections.java                             InputMismatchException.java              ↩
    ResourceBundle.java
Comparator.java                              InvalidPropertiesFormatException.java   Set. ↩
    java
ConcurrentModificationException.java         Iterator.java                            ↩
    SortedMap.java
Date.js                                      LinkedHashMap.java                       ↩
    SortedSet.java
Dictionary.java                              LinkedHashSet.java                      Stack. ↩
    java
DuplicateFormatFlagsException.java           LinkedList.java                          ↩
    StringTokenizer.java
EmptyStackException.java                     ListIterator.java                        ↩
    TooManyListenersException.java
Enumeration.java                             List.java                               TreeMap ↩
    .java
EventListener.java                           ListResourceBundle.java                 TreeSet ↩
    .java
EventListenerProxy.java                      Locale.java                              ↩
    UnknownFormatConversionException.java
EventObject.java                             MapEntry.java                            ↩
    UnknownFormatFlagsException.java
FormatFlagsConversionMismatchException.java  Map.java                                Vector. ↩
    java
FormatterClosedException.java                MissingFormatArgumentException.java      ↩
    WeakHashMap.java
HashMap.java                                 MissingFormatWidthException.java
```

5.6 Package java.util

TODO: how well supported are util.regex ?

```
/workspace/net.sf.j2s.java.core/src/java/util/regex> ls
Matcher.java  MatchResult.java  Pattern.java  PatternSyntaxException.java
```

Chapter 6

Integrated Java libraries

Java2Script is about reusing java code in our web pages and thus, it come with some libraries available so the java programmer can use familiar java tools. Some libraries available are: junit, swt, ajax/RPC, etc. The support for each of them and other tools will be described in this chapter.

6.1 Java2Script SWT

```
1. Eclipse SWT 3.1 API is ported for JavaScript in modern browsers,
Firefox, Internet Explorer, Opera, Safari
2. Desktop and window manager are built inside
3. Support UI blocking fow SWT's Shell and Dialog
4. Support SWT deffered layout
5. Support detecting font size change in Firefox
```

You may take a look at SWT example for a Java2Script application that intensively uses SWT. In this section however, we will make a SWT based application from scratch for showing how easily it is.

TODO: what is not supported is this version of SWT is the GC, etc.

6.2 Java2Script AJAX

TODO: do this section better.

```
1. HttpRequest/AJAX is introduced for both Java and JavaScript
2. Simple RPC API is introduced to call back Java server, which
supports Cross Site Scripting
3. Simple and Compound Pipe API is introduced to setup Comet
connection to server.
4. Support dynamic class loading for both Java and JavaScript
5. Support Java reflection (early implementation)
```

Chapter 7

Customizing output code with @j2s JavaDoc tags and annotations

Action may not bring happiness but there is no happiness without action.

—William James

The Java2Script compiler supports special syntax in JavaDoc comments for doing some interesting things at compile time. Java2Script extends the JavaDoc, and you can use the extended JavaDoc tags to do some jobs in JavaScript way. With the help of extended JavaDoc tag, we can add extra JavaScript codes or replacing the Java codes without modifying the Java codes.

Main reference page is http://j2s.sourceforge.net/articles/tutorial-advanced-programming-on-j2s.html

7.1 Using @j2s tags in JavaDoc

Java2Script compiler supports some "directives" that the user can include in its java code inside javadoc comments, that affects how a java element, like a method, a class or simple code block is translated to javascript. So the javadoc always must go before some java block of code. Also, usually this directives accepts some input that must be after the @j2s directive in the same javadoc. In the following java segment, we use the directive @j2sNative to tell the J2S compiler that a java code block must be replaced by some native javascript code that we give:

```
System.out.println("Code before");
/**
 * @j2sNative
 *
 * alert("a javascript native statement");
 */
{
    System.out.println("these statements will be replaced");
}
System.out.println("Code after");
```

that will be translated to

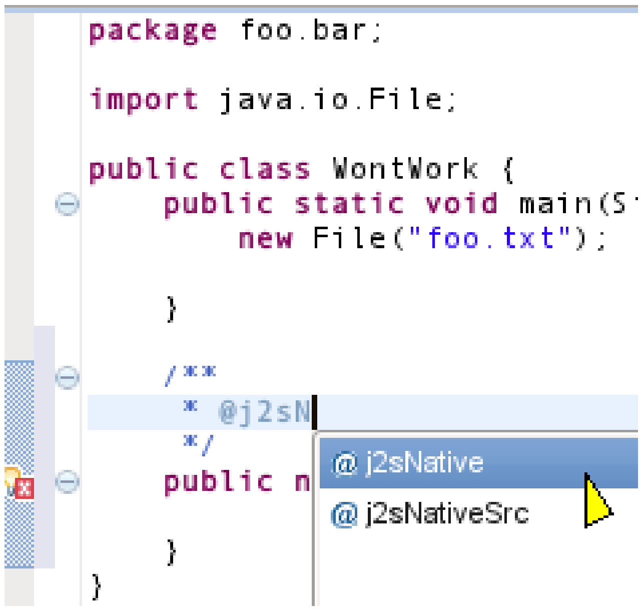
```
System.out.println("Code before");
{
alert("a javascript native statement");
}
System.out.println("Code after");
```

The important thing to notice here is how the java statement inside the block ("these statements will be replaced") is not present in the translated code, and it was replaced by the javascript statement `alert("a javascript native statement");` that we wrote inside the javadoc comment, after the @j2sNative directive.

All these @j2s annotations that we will describe later in more detail, are used the same way. First of all, they must be inside a JavaDoc comment. JavaDoc comments are special comments that begin with `/**` instead of `/*`. This JavaDoc comment have to go before a java code entity, like a method definition, a class definition or a code block. Second, the javadoc must contain a special J2S annotation like in the previous example, `@j2sNative` . We call this @j2s annotations *J2S compiler directives*, since they let us customize in some ways the compiler output code. Also, in most cases, the J2S compiler directive accepts some text input, like in the last example the javascript native code, and act over some java element, like in the last example, the java code block.

Tip

When you are inside a JavaDoc comment, if you type @j2s and then ctrl-space, eclipse will offer an autocompletion menu in which you can choose the desired @j2s annotation easily:



```
package foo.bar;

import java.io.File;

public class WontWork {
    public static void main(S
        new File("foo.txt");

    }

    /**
     * @j2sN
     */
    public n

    }
}
```

Figure 7.1: @j2s annotations eclipse auto completion menu

7.2 Writing native JavaScript code with @j2sNative

the @j2sNative is perhaps the more important @j2s directive of all. It allow the java programmer to replace a java block or method body with native javascript code.

Let's take simple example: you want to get current browser name in your java code. Java doesn't provide with such functionality, so we must to do it with native JavaScript code. (Remember that in javascript we can get the full browser name with the statement `navigator.userAgent`). For doing this we will use the @j2sNative JavaDoc annotation for embbeding native javascript code inside the java code. In the following example, notice the java comment with the `@j2sNative` string and the javascript code before the empty java block `{}`:

```
String fullBrowserName = null;❶

/**❷
 * @j2sNative❸
 * fullBrowserName = navigator.userAgent; ❹
 */{}❺

System.out.println("Full browser name is : "+fullBrowserName);
```

1. Define a java String variable named `fullBrowserName`for storing the Browser name
2. Here starts JavaDoc that will be use for embbeding native javascript in our java code
3. we use the annotation @j2sNative for indicating to the Java2Script compiler that the next java block must be replaced with þe javascript code that follows this annotation (`fullBrowserName = navigator.userAgent;`).
4. This is the native javascript code that will replace the next java block of code. Notice that the "*" characters in the javadoc are ommited and that the native code goes from after the @j2sNative annotation to the end of the JavaDoc. Also notice that in this case we are assigning the value of the native string `navigator.userAgent` to the java variable `fullBrowserName`
5. This is the java code block that the @j2sNative directy will apply to. The body of this java block will be replaced by the native javascript code inside the javadoc.

The above code will be translated to the following javascript code:

```
var fullBrowserName = null;
{fullBrowserName = navigator.userAgent;}
System.out.println ("Full browser name is : " + fullBrowserName);
```

Notice how the empty java code block has been filled with the native javascript code that we introduced inside the @j2sNative JavaDoc.

Since this compiler directive is so important and has several usage scenarios, we will discuss its usage in detail in a separate chapter Chapter 9.

A more complex example that shows all use cases. Read the javadocs for info about each of them.

```
package net.sf.j2s.tutorial.debug;

public class HelloJ2SNative {
  /**
   * For java native methods, if Java2Script native codes are given, the
   * native method will be generated when Java2Script compiler is enabled
   *
   * @j2sNative
   * var styleCSS = "position:absolute;left:150px;top:5px;width:100px;height:40px;"
   *    + "text-align:center;font-weight:bold;color:yellow;background-color:blue;"
   *    + "border:1px solid red;";
   * var hiEl = document.createElement("<div style=\"" +  styleCSS + "\"></div>");
   * document.body.appendChild(hiEl);
   * hiEl.appendChild (document.createTextNode ("Hi!"));
   */
  native static void sayHi();

  /**
   * For java native methods, if you did not define Java2Script native codes,
   * the method will be simply ignored when generating the JavaScript codes.
   */
  native static void sayWell();

  /**
   * When j2sNative is defined fro a method, the method body will be overrided
   * with the given native JavaScript codes.
   *
   * @j2sNative alert ("Hello, JavaScript");
   */
  static void sayHello() {
    System.out.println("Hello, Java");
  }

  static void sayHei(String name) {
    if(name != null)
    /**
     * For a normal block, inert j2sNative Javadoc before the brace will
     * replace the inner block body.
     *
     * @j2sNative alert ("Hei, " + name);
     */
    {
      System.out.println("Hei, " + name + ", how are you?");
    } else
    /**
     * @j2sNative alert ("Hei.");
     */
    {
      System.out.println("Hei, how are you?");
    }
  }

  static void saySomething() {
    /**
     * Insert JavaScript only codes with an empty brace block.
     *
     * @j2sNative alert ("En, things will go fine.");
     */
    {}
  }

  public static void main(String[] args) {
    /*
     * When running from Java, the following line must be commented, as
     * there are not implemented native codes for the native method #sayHi
     */
    sayHi();
    sayHello();
    sayHei("Josson");
    saySomething();
  }
}
```

And the following is the generated javascript code. We indicate mark with 2 new lines where the native code embbeded with @j2sNative is for each directive call.

```
Clazz.declarePackage ("net.sf.j2s.tutorial.debug");
c$ = Clazz.decorateAsClass (function () {
Clazz.instantialize (this, arguments);
}, net.sf.j2s.tutorial.debug, "HelloJ2SNative");
c$.sayHi = Clazz.defineMethod (c$, "sayHi",

function () {
var styleCSS = "position:absolute;left:150px;top:5px;width:100px;height:40px;"
+ "text-align:center;font-weight:bold;color:yellow;background-color:blue;"
+ "border:1px solid red;";
var hiEl = document.createElement("<div style=\"" +  styleCSS + "\"></div>");
document.body.appendChild(hiEl);
hiEl.appendChild (document.createTextNode ("Hi!"));
});
c$.sayHello = Clazz.defineMethod (c$, "sayHello",

function () {
alert ("Hello, JavaScript");
});
c$.sayHei = Clazz.defineMethod (c$, "sayHei",
function (name) {
if (name != null) {

alert ("Hei, " + name);
} else {

alert ("Hei.");
}}, "String");
c$.saySomething = Clazz.defineMethod (c$, "saySomething",
function () {
{

alert ("En, things will go fine.");
}});
c$.main = Clazz.defineMethod (c$, "main",
function (args) {
net.sf.j2s.tutorial.debug.HelloJ2SNative.sayHi ();
net.sf.j2s.tutorial.debug.HelloJ2SNative.sayHello ();
net.sf.j2s.tutorial.debug.HelloJ2SNative.sayHei ("Josson");
net.sf.j2s.tutorial.debug.HelloJ2SNative.saySomething ();
}, "Array");
```

7.3 Using real Java annotations instead JavaDoc for @j2s

for each @j2s javadoc tag there is an equivalent java annotation that we can use. In some cases, the java @J2S annotation can be more convinient than the javadoc comment.

you must include net.sf.j2s.ajax project in yout java classpath fgor using them... TODO: better this section

J2s also support using real java annotations (not JavaDoc) instead @j2s annotations inside JavaDoc, For each @j2s JavaDoc annotation there exists a @J2S java annotation class that can be used:

```
  If you want to refactoring use @J2SRequireImport annotation:
/**
 */
@J2SIgnoreImport(B.class)
public class A {
}
instead of javaDoc
/**
 * @j2sIgnoreImport("a.b.c.B")
 */
public class A {
}

All @j2s* javaDoc has its @J2S* annotations.
```

Chapter 8

Code Translation

"The question of whether computers can think is like the question of whether submarines can swim."

—Edsger W. Dijkstra

Now that the user knows the basics of using the Java2Script plugin, int this chapter we will examine the generated code and how Java2Script translate java sources to javascript and how we can use this code in our html documents for lowding our translated java applications.

Note that you won't normally need to understand how the compiler javascript code work, since one of the main idea of Java2Script plugin is to abstract the java user from javascript code at all. Nevertheless, it is generally a good idea to be familiar on how J2S generated code works, but of course you can choose to jump to next section.

A java to javascript code translator like J2S must translate each possible java statement to its javascript equivalent. There are some constructions that are very similar in both languages like, `for, if, while, do`, and other expressions. But there are Java language constructions like class, method, interface, that are not present in javascript. So the compiler has also to add artificial support for these.

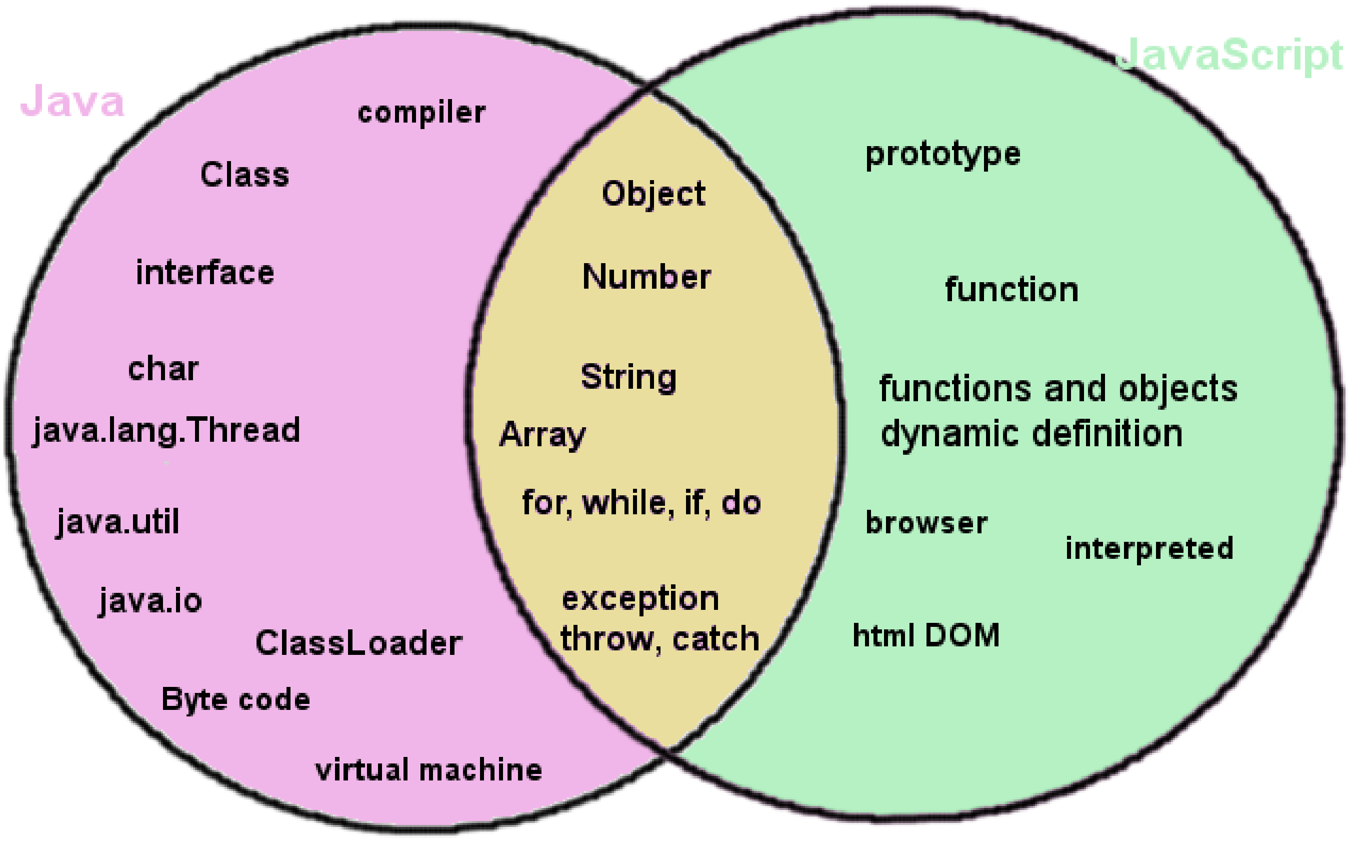


Figure 8.1: Concepts from both languages and what J2S takes from both worlds

8.1 Generated JavaScript files

In this section we will examine the generated JavaScript files and also the generated HTML document for a basic understanding of how java code is translated to JavaScript and how that JavaScript can be loaded and executed from an HTML document.

Consider the following simple java class that contains one static method and one instance method:

```
package my.first.project;
public class HelloWorld {
  public String getName() {
    return this.getClass().getName() + this.hashCode();
  }
  public static void main(String[] args) {
    HelloWorld obj = new HelloWorld();
    System.out.println("object name " + obj.getName());
  }
}
```

And the generated javascript, with comments, is the following:

```
Clazz.declarePackage ("my.first.project");❶
c$ = Clazz.declareType (my.first.project, "HelloWorld");❷
Clazz.defineMethod (c$, "getName", function () {❸
  return this.getClass().getName() + this.hashCode();
});
```

```
c$.main = Clazz.defineMethod (c$, "main", function (args) {❹
  var obj =  new my.first.project.HelloWorld ();
  System.out.println ("object name " + obj.getName ());
}, "~A");
```

❶ this is the translation of the java statement `package my.first.project;` , declaring the package my.first.project. As with java, it is required before a class definition.

❷ This is the translation of the java code `public class HelloWorld`. With `Clazz.declareType (my.first.project, "HelloWorld")` we declare the new class `HelloWorld` of package `my.first.project` an it returns the new created class object that we later can use for adding methods and such things.

❸ This is the translation of the java code `public class HelloWorld`. With `Clazz.declareType (my.first.project, "HelloWorld")` we declare the new class `HelloWorld` of package `my.first.project` an it returns the new created class object that we later can use for adding methods and such things.

❹ this is how the static method main is defined in javascript. Note that we pass the following parameters to `Clazz.defineMethod`: class object (returned by declareType), method name, method function.

also notice that the only difference between declaring a static and instance methods is that in the case of static method declaration we assign the returned method to a class object property, i.e., `$c.main = ...`

As we can see, the `Clazz` object contains functions that emulate the java language. We will document all this functions in Section A.1.

Other thing to notice is that the code translation is linear: for a java statement there is a single javascript statement. Also, it is noticeable that, for the point of view of the translator, there are two main types of Java statements:

- **Supported by javascript** Most java statements, like method calling, for, while, if, etc. expressions, are supported directly by the javascript language and do not need almost any translation. In the previous example, the java statement `System.out.println ("hello world");` remains the same both in java and in javascript.
- **Containing java concepts not supported by javascript.** Doing Object Oriented programming stuf like cl'asses, methods definition, etc. Here is when we use the object Clazz: the J2S java language emulator for javascript. So, for example, if we ewant to define a new class in javascript, we use Clazz.declareType.

Now let's finnish examining a more advance java example and its transjaltion to javascript:

```
package my.first.project;

import java.util.HashMap;
import java.util.Iterator;

public class MyTable extends HashMap<String, Integer>{
  public String print() {
    String s = "";
    Iterator<String> i = keySet().iterator();
    while (i.hasNext()) {
      s += i.next()+",";
    }
    return s;
  }
}
```

that is translated to the following commented javascript code:

```

Clazz.declarePackage ("my.first.project");
Clazz.load (["java.util.HashMap"], "my.first.project.MyTable", null, function () {❶
  c$ = Clazz.declareType (my.first.project, "MyTable", java.util.HashMap);❷
  Clazz.defineMethod (c$, "print", function () {❸
    var s = "";
    var i = this.keySet ().iterator ();
    while (i.hasNext ()) {
      s += i.next () + ",";
    }
    return s;
  });
});
```

❶ This is different that our previous example. Since in this case, our class extends java.util.HashMap, we need first of all, that java.util.HashMap to be loaded and only then declare our new type. The function `Clazz.load` help us on this. Note that the actual call to `Clazz.declareType` is inside the callback function passed to `Clazz.load`.

❷ At this point the required class java.util.HashMap is loaded and so we can call Clazz.declareType. Notice the last argument `java.util.HashMap` referencing the parent class. Note that in javascript we can reference java Class objects just like in java.

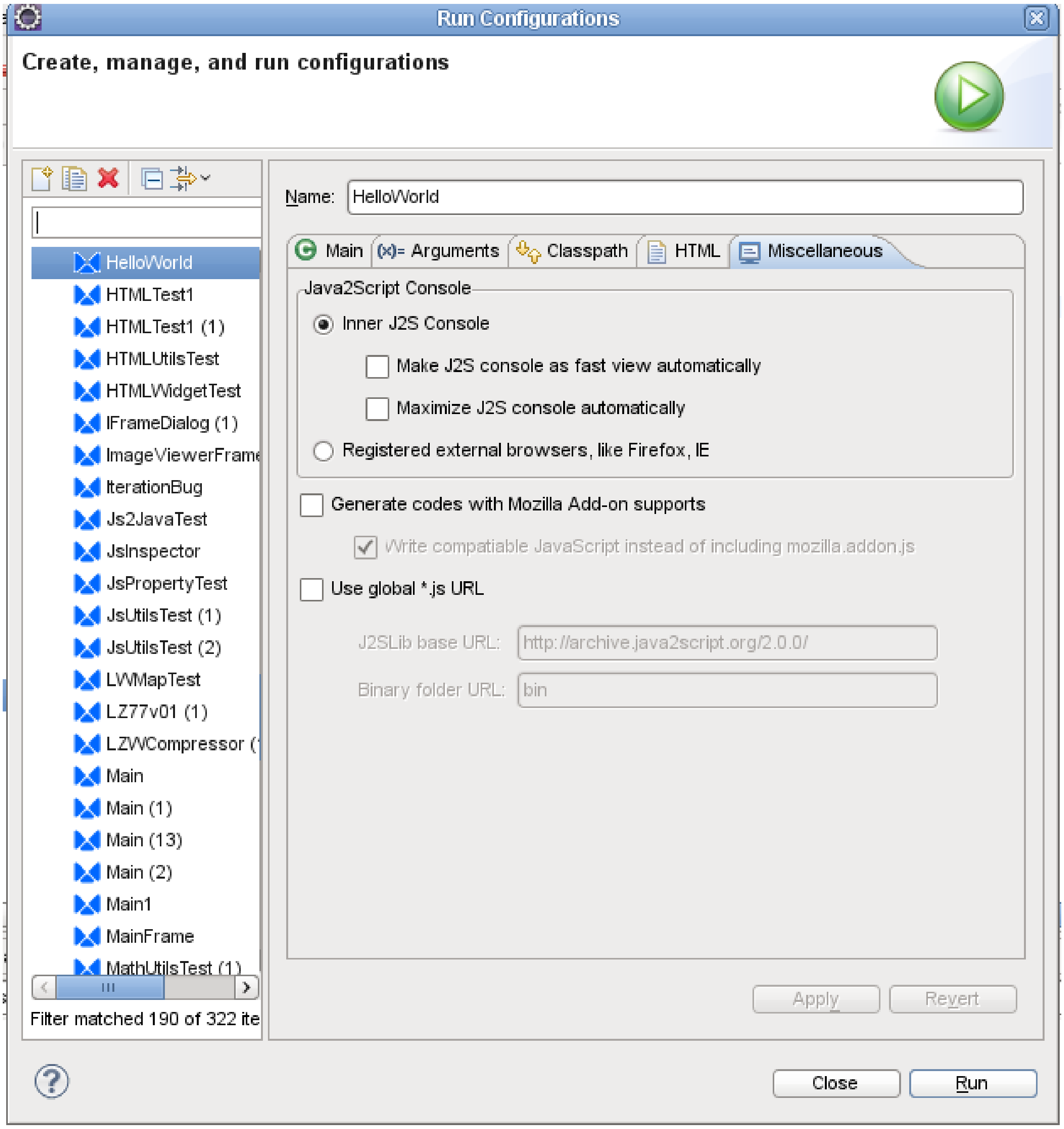
❸ Notice that calling `this.keySet()` is done just like in java, with the exception that in javascript the `this` keyword is not optional.

8.2 Doing java in HTML documents

When you run a J2s application in eclipse with menu Run->Run as..->Java2Script Application the Java2Script plugin generates a simple html document that execute the java application. For doing that it is required to perform at least the following tasks:

- load the j2slib.z.js Java2Script runtime script
- configure J2S runtime
- Load main class java class dependencies
- execute some ava code like calling the main() method of main class.

You can use the generated html file in yout web applications, nevertheless, it is very common to launch your Java2Script application from an existing html document, or from a server page (php, jsp asp, whatever). In those cases, it is very helpful to understand how J2s applcations are loaded and executed. So lets examine a minimal html document that calls our HelloWorld java example. For generating the following html markup, the HelloWorkd java class is launched with a clean configuration:



Name: HelloWorld

Java2Script Console

(●) Inner J2S Console

[ ] Make J2S console as fast view automatically

[ ] Maximize J2S console automatically

( ) Registered external browsers, like Firefox, IE

[ ] Generate codes with Mozilla Add-on supports

[✓] Write compatiable JavaScript instead of including mozilla.addon.js

[ ] Use global *.js URL

J2SLib base URL: http://archive.java2script.org/2.0.0/

Binary folder URL: bin

Figure 8.2: A clean J2S Application launcher configuration

A short version of generated HTML document is the following:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/ ↩
    xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head><title>my.first.project.HelloWorld</title>
```

```
<script type="text/javascript" src="../path/to/j2slib/j2slib.z.js"></script>❶
</head>
<body>
  <script type="text/javascript">
    ClazzLoader.packageClasspath ("java", "../path/to/j2slib/", true);❷
    ClazzLoader.setPrimaryFolder ("bin");❸
    ClazzLoader.packageClasspath (["my.first.project", "net.sf.j2s.probes"], "bin/");  ❹
    ClazzLoader.loadClass ("my.first.project.HelloWorld", function () {❺
      my.first.project.HelloWorld.main([]);❻
    });
  </script>
</body>
</html>
```

❶ First of all it is required to include the script j2slib.z.js.

❷ With this call to `ClazzLoader.packageClasspath` we tell the java VM where are main java sources like java.lang, java.util, etc. classes, usually at the same location of file j2slib.z.js.

❸ Calling `ClazzLoader.setPrimaryFolder` indicates to the java VM what is the primary folder. In the case of a java package collition, the primary folder has precedence.

❹ Exactly the same way we have done for java main sources, this call tells the VM where to find the classes of our project, and that is at the "bin" folder.

❺ Now the intersting part. At this point we are ready to invoke some java code from our html document. We have already tell the VM where are the sources of each java package we need. Remember that in Java2Script will lazily load all java classes (load only when needed). So, for executing some java code that involves some java classes, we first need to load some java classes and only when loaded, we can do our java. In this case, we need to invike HelloWorld.main(), and so, we ask to load the class my.first.project.HelloWorld and only when it is loaded (in the function callback, second parameter), we invoke our java code `my.first.project.HelloWorld.main([]);`.

❻ As we said in the previous item, this is the actual java code that will be invoked. Note that the java sintax is very straightforward. Also notice that in this case we only need to call main method, but nothing prevent us for doing more complex things, like:

```
var requiredClasses = ["my.first.project.HelloWorld",
  "new java.util.HashMap", "java.util.Arrays"];
ClazzLoader.loadClass (requiredClasses, function () {
  var arr = ["hello", "world", "1", "two"];
  var map1 =  new java.util.HashMap ();
  for (var i = 0; i < arr.length; i++) {
    map1.put (new Integer (arr[i].length), arr[i]);
  }
  java.util.Arrays.sort(arr);
});
```

As we can see, the ClazzLoader javascript object contains an API do dialogate with the java runtime from javascript. It lets us configure and manipulate the "java virtual machine", for example, telling where main java library code is located, set the "class path", dynamically loading java classes and give as a context from where to call java classes

We will examine and document each of this functions in Section A.2.

8.3 Types

Explain how j2s emulates java types. ej: string and characters are all javascript strings, long, int short, double, float are all javascript numbers. arrays are javascript arrays

Java2Script also wrapps some exceptions that can occurr

8.4 Exceptions

Exceptions are well supported by Java2Script and so exception related statements have a very simple translation. There is only one special thing we have take care about when programming Java2Script code and is that in Java2Script, besides Java exceptions, there are also JavaScript native exceptions. For example, as we have seen, using @j2sNative compiler directive we can introduce native javascript code. And native javascript code can throw native javascript exceptions, that may have not be related at all with java exceptions (java.lang.Exception class). For example, the following example will show native javascript code that throws a native DOM exception.

```
/**
 * native DOM exception siumulation
 * @j2sNative
 * document.body.appendChild(null);
 */{}
```

It is important to understand that these native exceptions, are not java.lang.Exception instances and so, they cannot be catched with `catch(Exception e)` expression. So, how we catch native exceptions and how can we diferentiate them from common java exceptions? The answer is that native exceptions are instance of class java.lang.Throwable (a superclass of java.lang.Exception) and so, we can catch them using Throwable, as the following example shows:

```
try {
  /**
   * native DOM exception siumulation
   * @j2sNative
   * document.body.appendChild(null);
   */{}
} catch (Exception e) {
  //catching java exceptions
  System.out.println("this is NOT printed!!!");
}
catch(Throwable e) {
  //catching native exceptions!
  System.out.println("this is printed!!!");
}
```

The only special case of native exceptions are the null pointer exceptions. Since both in java and javascript null is the same object, and in java we spect NullPointerException to be throwed when accessing the null object, native javascript null pointer exceptions will be objects of the class java.lag.NullPointerException. Let understand this with a litlle example. In the following listing, we emulate a native null pointer exception and show that is catched with a NullPointerException and not as another native exception with Throwable:

```
   try {
      /**
       * This simulates a native null pointer exception
       * @j2sNative
       * var a = null;
       * a.sdf();
       */{}
    } catch (NullPointerException e) {
      System.out.println("printed!");
    }
    catch (Throwable e) {
      System.out.println("NOT printed!");
    }
```

Chapter 9

Working With native code

in section we have explained how to use J2S compiler directives like @j2sNative inside javadocs for customizing compiler output code, and specially the directive @j2sNative for including native javascript in our java classes.

Nevertheless, there are some areas that need to be understand if you want to write native javascript code and need to integrate that native code with your java code.

this chapter will describe how, call java methods from javascript code, problems with java object+method vs - javascript objectsand javascript functions, care native objects like dom, etc,

TODO

j2s-native-mode support is expl here

9.1 Native Objects

a native object is TODO

why we have to be careful when dealing with native objects?

TODO

9.2 Accessing native JavaScript objects from Java

In section Writing native JavaScript code with @j2sNative we learn how to include our JavaScript code in our Java classes and how our native JavaScript code can call our Java objects. But, wait a moment, in JavaScript we have access to a the document object model, and a lot of JavaScript utilities and toolkits .... TODO

In Java, all objects are inherited from Java.lang.Object. So they must implement methods like hashCode(), equals(), etc. We say that an object is native when it is created from native JavaScript code and doesn't implements java.lang.Object. .... TODO

9.3 Native objects mode support

By default, J2S will overwrite javascript Object.prototype adding methods of java.lang.Object like equals, hashCode, notifyAll, etc, so all created objects are valid java objects. Normally this is fine because you don't have to worry about javascript objects that are not valid java objects.

Nevertheless, in some situations, this default behaviour is not desiderable. For example, if you want to integrate your J2S application with a 3rd party javascript toolkit, there can be compatibilities issues because some javascript toolkits simply don't work with a modified Object.prototype. More, since all javascript objects contains java.lang.Object methods, it is impossible to create a clean javascript object, something really common and required for working with 3rd party javascript toolkits.

Fortunately J2S supports a "native object" mode, in which javascript Object.prototype is not contaminated

explain the problems with native objects and explain j2s.native mode

9.4 JavaScript functions

TODO: this sections contains the escense but has a bad english. it should be rewritten more elegantly.

The first problem that we face when working with native javascript is that , in javascript, we work with functions and the java language doesn't support such a concept or nothing similar. In java, the only beahaviour language concepts we have are instance methods and static methods. Mixing native javascript functions and java code without realizing what we really are doing, can cause very nasty bugs and problems, and so in this section we will examine possible problems and advices about how to do it right.

function context:

what is a function context? In javascript, any function can be evaluated using any object as its context. The context of a function when evaluated is the object referenced by the "this" keyword.

Important
Javascript function can be called with an arbitrary context object while Java instance methods must always have a fixed context.

since in java we always have to use an object, the function in java have a fixed context (the this object). But in javascript this is not true and we must ensure the context. for example, the following example for registering a mouse click listener in document.body will work, but it is incorrect! :

```
Runnable r = new Runnable(){
  public void run() {
    System.out.println("clicked");
  }
};
/**
 * @j2sNative
 * document.body.onclick=r.run; //WRONG!
 */{}
```

Why this is wrong? When an html document is clicked, and a function is registered for listen to clicks with "element.onclick", the function will be evaluated using the clicked html element as the function context. This means that any reference to "this" in the function body, will point to the target html element and not where it should be, to the Runnable instance. Let examine an example that fails because of this:

With java static methods there is no problem because they do no context (no this pointer). But it is not useful to work with static methods for

Chapter 10

Client - Server communication

"It is impossible for a man to learn what he thinks he already knows."

—Epictetus

this chapter explain different techniques and tools related to client-server comunicacion in J2S applications. TODO

10.1 ajax

TODO

10.2 Simple RPC

http://inside.java2script.com/2007/06/02/tutorial-of-java2script-swt-and-simple-rpc-application.html

10.3 Simple pipe / commet

simple pipe : http://inside.java2script.com/2007/08/01/introducing-java2scripts-simple-pipe.html

http://inside.java2script.com/category/comet

Chapter 11

Getting started with Java2Script sources

"What is happiness? The feeling that power is growing, that resistance is overcome."

—Friedrich Nietzsche

11.1 Getting Java2Script sources

This section is for those developers who want to install Java2Script sources, run the plugin in Eclipse plugins sdk for testing, see how J2S works, be able to change it, fix it, redistribute it with your own modifications, etc

We will use Eclipse 3.6. You should choose an Eclipse distribution that supports for Eclipse plugin development (almost all).

11.1.1 Install Eclipse 3.6 and Java2Script plugin

We will need Eclipse 3.6 with the Java2Script plugin installed in order to build sources. [1]

If you already have Eclipse 3.6 installed and are comfortable doing your Java programming there, you can use it. If not, download an Eclipse 3.6 distribution oriented to Java and Eclipse plugin development, like Eclipse 3.6 Classic. Also download Java2Script plugin for Eclipse 3.6 from Java2Script homepage

As explained in previous sections, installing Java2Script plugin is easy: just extract files and overwrite the `plugins` Eclipse folder content and restart Eclipse.

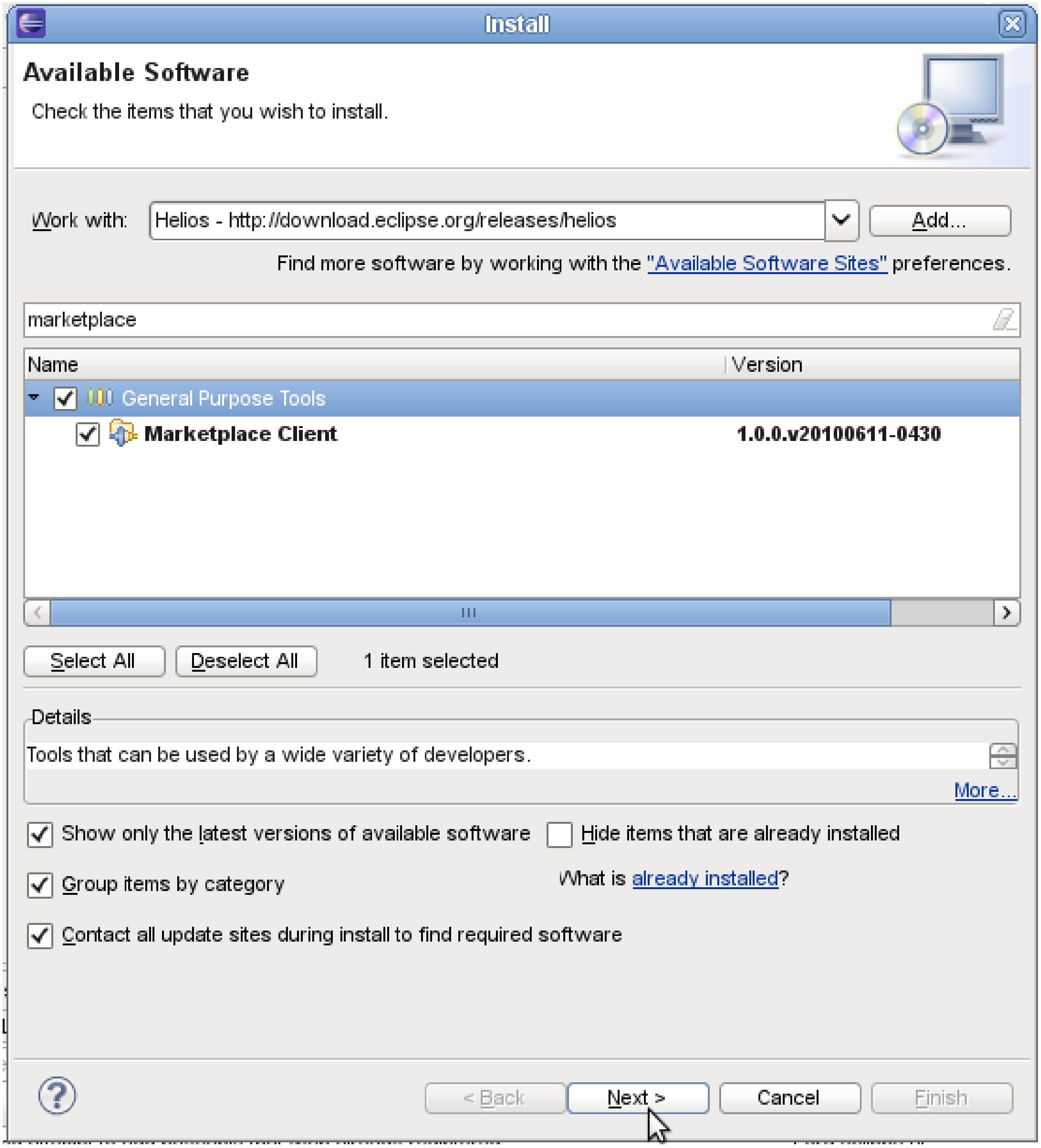
Once Eclipse has been restarted, you can verify whether Java2Script plugin is successfully installed or not by going to

11.1.2 Install SVN support

If you are a developer you probably already has SVN support in your Eclipse. If not, this section will explain how you can easily install subclipse using Eclipse marketplace.

Check if you have the Help->Eclipse Marketplace menu item. If so please go to the next paragraph. If not, we will first need to install the Eclipse marketplace client. It is easy, just go to menu `Help->Install new Software...` In the dialog, enter "Work with" `Helios - http://download.eclipse.org/releases/helios`, wait a moment to Eclipse to download the available software catalog and then search the work "marketplace" as shown in the following figure. Check "Marketplace Client", follow the steps and wait until Eclipse marketplace client is downloaded and installed:

[1] Yes, Java2Script sources are compiled using the Java2Script compiler! For example, many .java files, like java.lang, java.util, org.eclipse.swt, etc need to be compiled to JavaScript in order to build the j2slib JavaScript library which all Java2Script web applications are based on.


Install
Available Software
Check the items that you wish to install.
Work with:
Add...
Find more software by working with the "Available Software Sites" preferences.
Name
Version
Select All
Deselect All
Details
More...
What is already installed?
< Back
Next >
Cancel
Finish

Work with: Helios - http://download.eclipse.org/releases/helios

Filter: marketplace

Name	Version
✓ General Purpose Tools	
✓ Marketplace Client	1.0.0.v20100611-0430

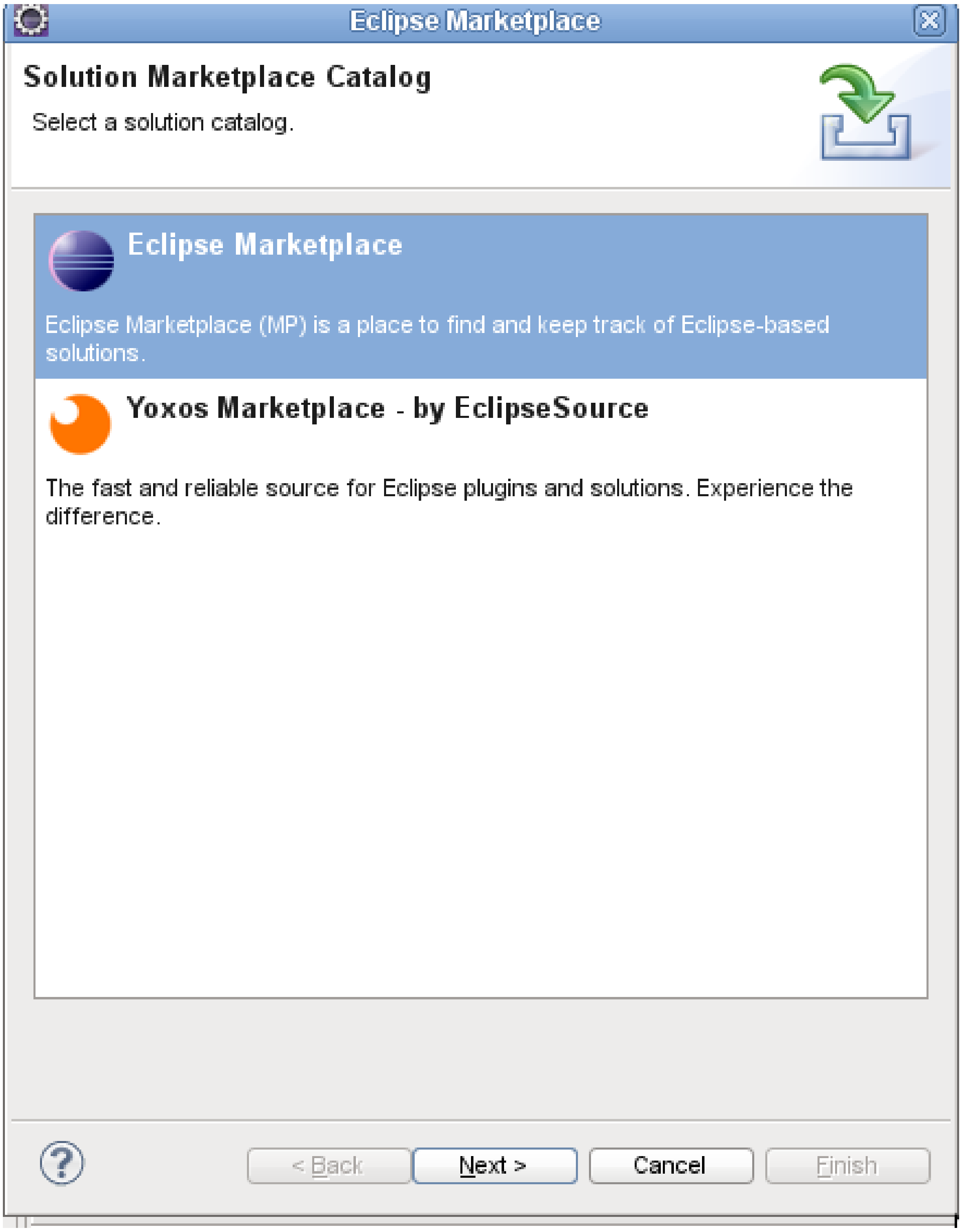
1 item selected

Details: Tools that can be used by a wide variety of developers.

- ✓ Show only the latest versions of available software
- ☐ Hide items that are already installed
- ✓ Group items by category
- ✓ Contact all update sites during install to find required software

Figure 11.1: Installing Eclipse Marketplace client

At this point you should have Eclipse marketplace client support in your Eclipse installation. So, we will install subclipse plugin for SVN team support. Goto Help->Eclipse Marketplace, choose "Eclipse marketplace" and press Next



Solution Marketplace Catalog

Select a solution catalog.

Eclipse Marketplace

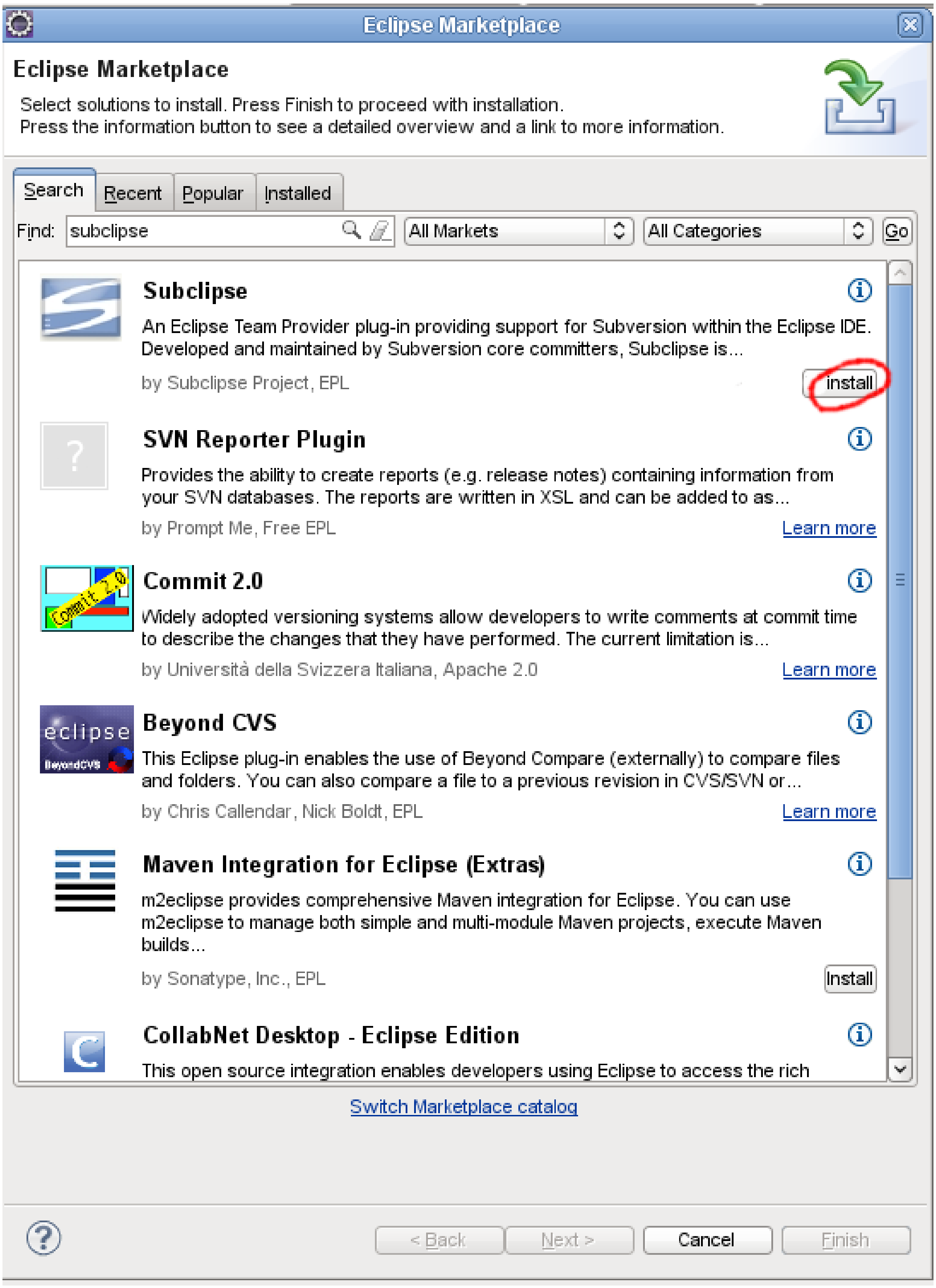
Eclipse Marketplace (MP) is a place to find and keep track of Eclipse-based solutions.

Yoxos Marketplace - by EclipseSource

The fast and reliable source for Eclipse plugins and solutions. Experience the difference.

Figure 11.2: Open Eclipse Marketplace

Click on next and search "subclipse" and then click on "install" button:



Eclipse Marketplace

Select solutions to install. Press Finish to proceed with installation.
Press the information button to see a detailed overview and a link to more information.

Find: subclipse

Subclipse
An Eclipse Team Provider plug-in providing support for Subversion within the Eclipse IDE. Developed and maintained by Subversion core committers, Subclipse is...
by Subclipse Project, EPL

SVN Reporter Plugin
Provides the ability to create reports (e.g. release notes) containing information from your SVN databases. The reports are written in XSL and can be added to as...
by Prompt Me, Free EPL

Commit 2.0
Widely adopted versioning systems allow developers to write comments at commit time to describe the changes that they have performed. The current limitation is...
by Università della Svizzera Italiana, Apache 2.0

Beyond CVS
This Eclipse plug-in enables the use of Beyond Compare (externally) to compare files and folders. You can also compare a file to a previous revision in CVS/SVN or...
by Chris Callendar, Nick Boldt, EPL

Maven Integration for Eclipse (Extras)
m2eclipse provides comprehensive Maven integration for Eclipse. You can use m2eclipse to manage both simple and multi-module Maven projects, execute Maven builds...
by Sonatype, Inc., EPL

CollabNet Desktop - Eclipse Edition
This open source integration enables developers using Eclipse to access the rich

Switch Marketplace catalog

Figure 11.3: search for subclipse

TODO At last, we must restart Eclipse to apply the changes.

11.1.3 Import Java2Script sources

Now that we have SVN support in our Eclipse, we can check out Java2Script source code. Go to File->Import...->SVN->Checkout projects from SVN.

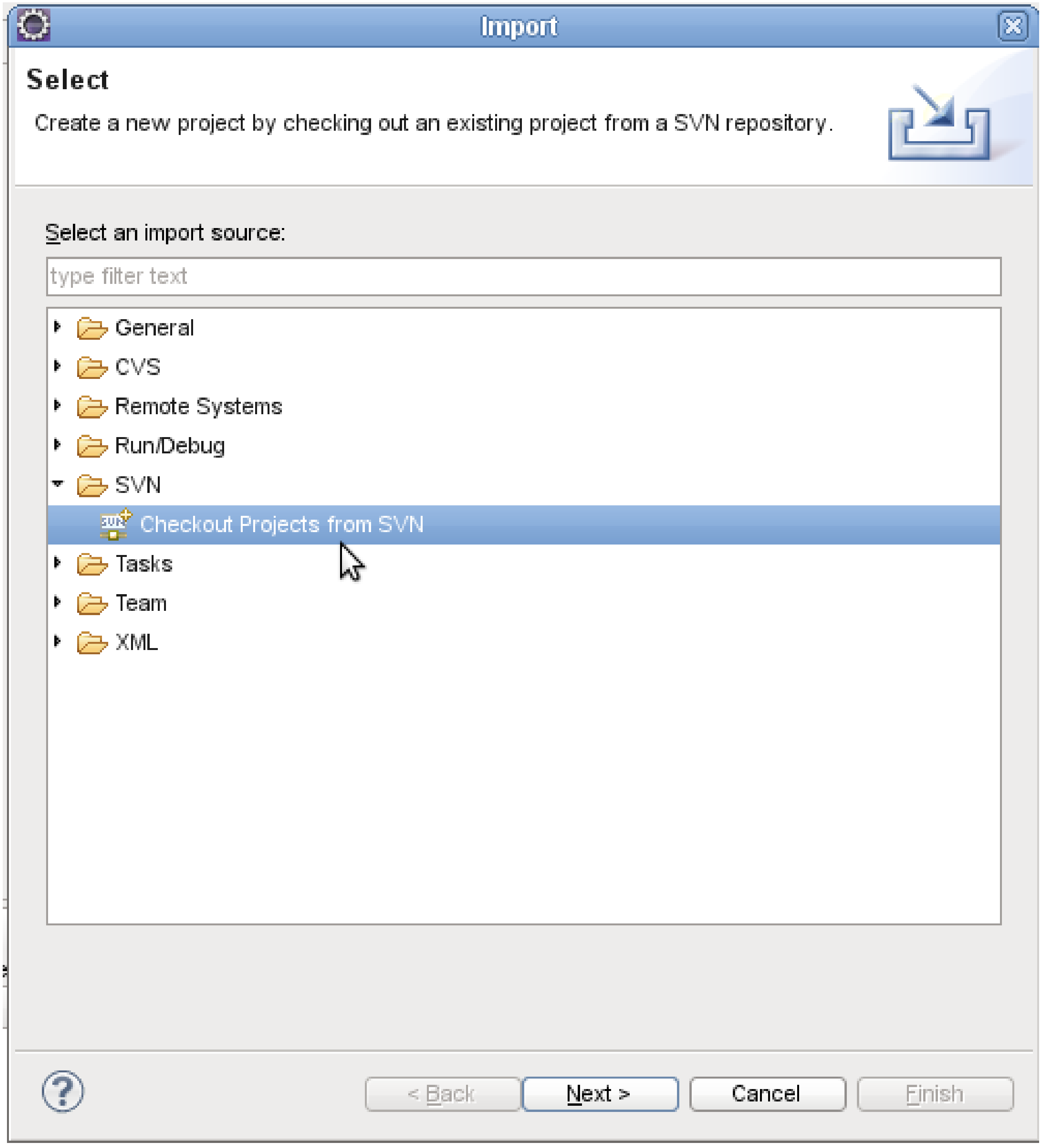


Figure 11.4: Import sources from SVN repository

Select "Create a new Repository location", click in "next" and enter `https://java2script.googlecode.com/svn-/trunk/` [2]. Clicking on next will load all folders of the project Java2Script. Expand folder "sources", and select all its child folders except `net.sf.j2s.core` and `net.sf.j2s.ui`:

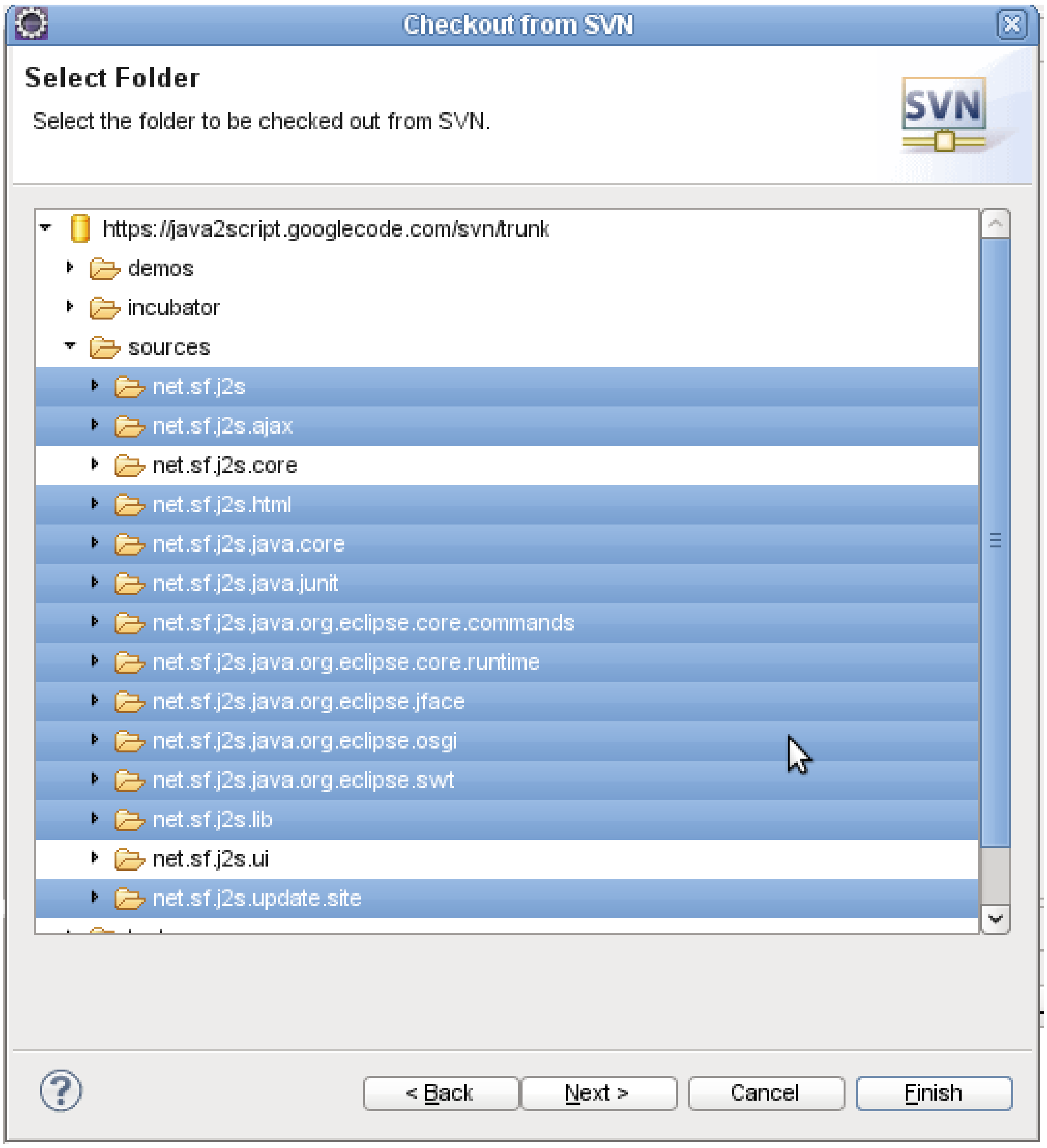

Checkout from SVN
Select Folder
Select the folder to be checked out from SVN.
SVN
https://java2script.googlecode.com/svn/trunk
demos
incubator
sources
net.sf.j2s
net.sf.j2s.ajax
net.sf.j2s.core
net.sf.j2s.html
net.sf.j2s.java.core
net.sf.j2s.java.junit
net.sf.j2s.java.org.eclipse.core.commands
net.sf.j2s.java.org.eclipse.core.runtime
net.sf.j2s.java.org.eclipse.jface
net.sf.j2s.java.org.eclipse.osgi
net.sf.j2s.java.org.eclipse.swt
net.sf.j2s.lib
net.sf.j2s.ui
net.sf.j2s.update.site
< Back
Next >
Cancel
Finish

Figure 11.5: Import sources from SVN repository

[2]We take this url from Java2Script googlecode site.

Click on finnish for start checking out Java2Script project sources.

Because we want to support for Eclipse 3.6 we have to obtain the sources of `net.sf.j2s.core` and `net.sf.j2s.ui` from another location: `https://java2script.googlecode.com/svn/branches`. So we repeat the process for that repository location, selecting `net.sf.j2s.core_3.6` and `net.sf.j2s.ui_3.6` folders:

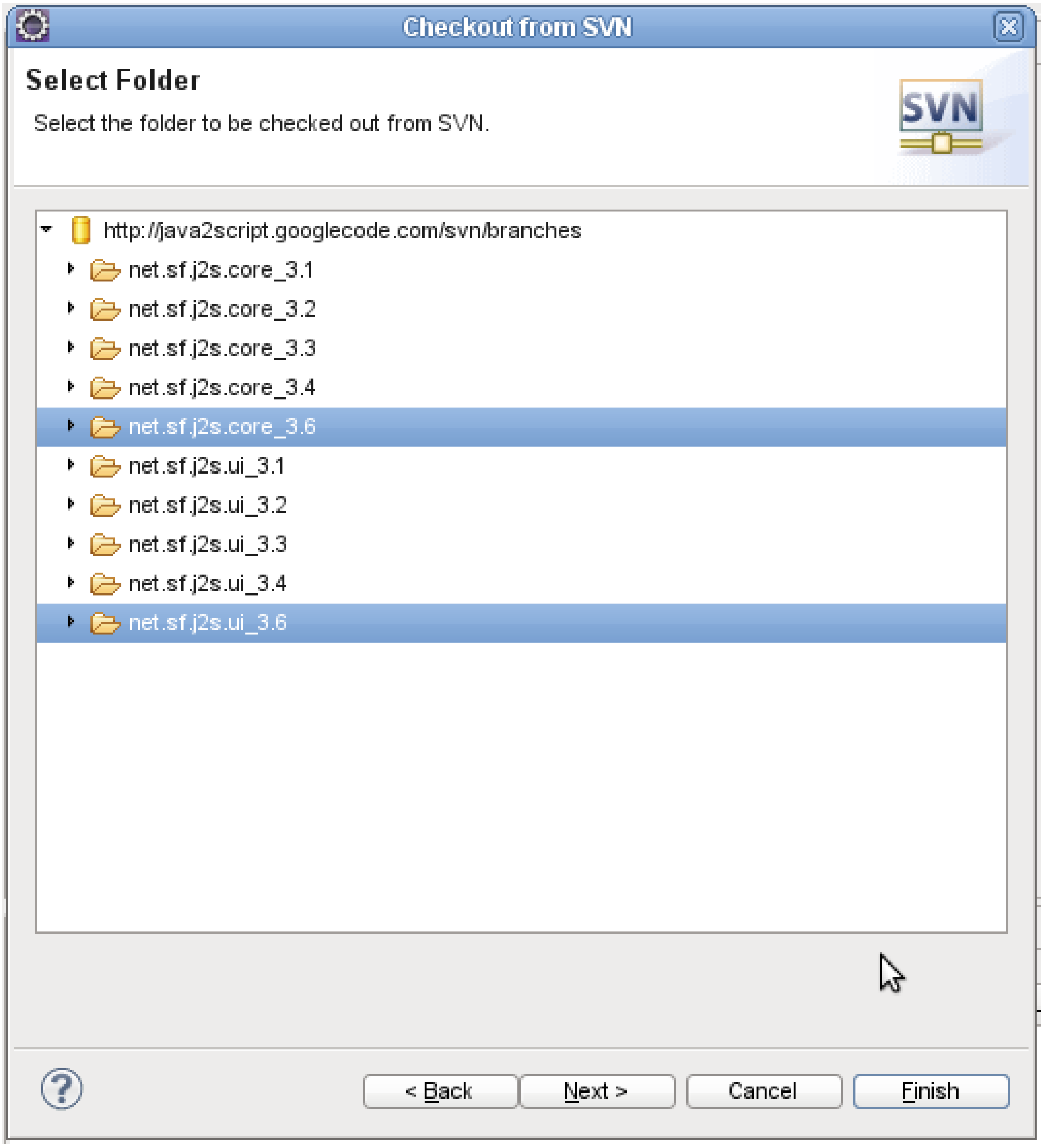
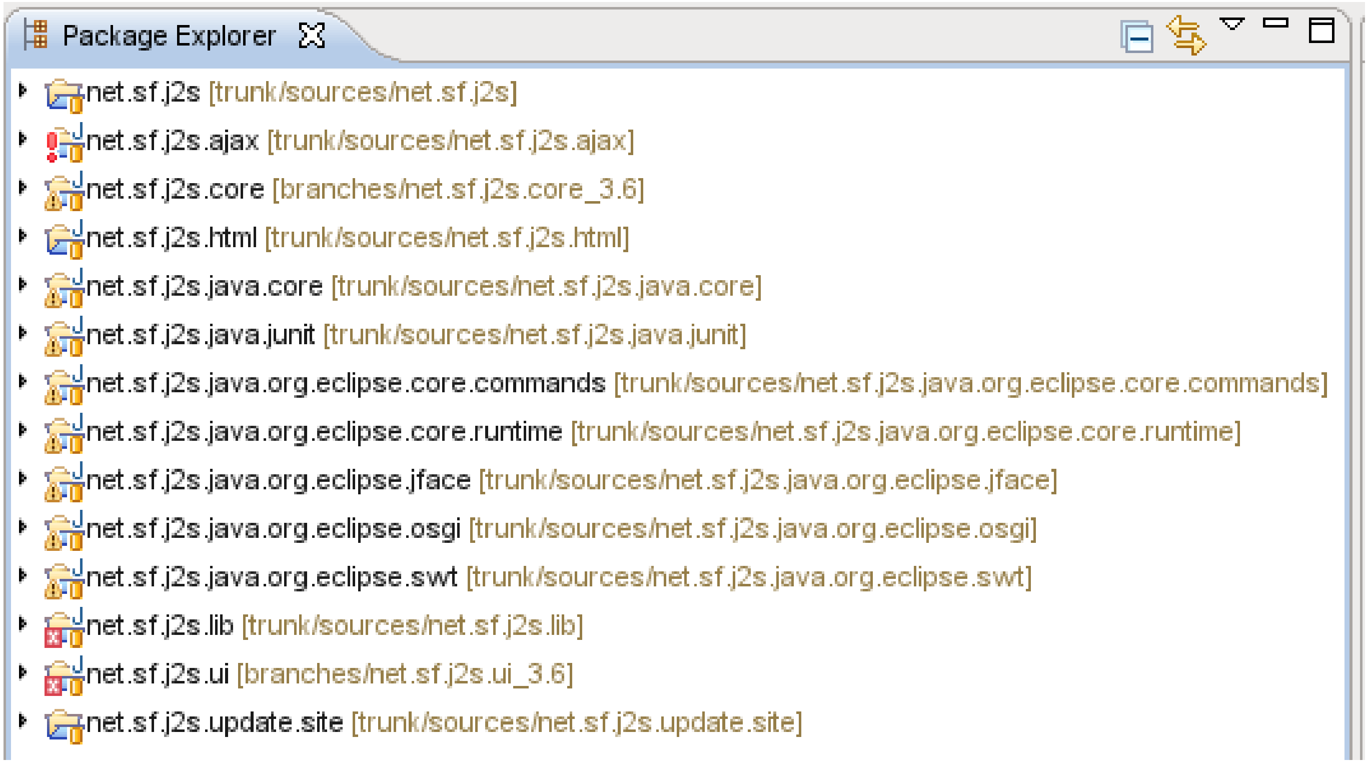


Figure 11.6: Import sources from SVN repository

After this, the lastest codes are checked out. You shoud see all Java2Script related projects in your Eclipse workspace :


Package Explorer

net.sf.j2s [trunk/sources/net.sf.j2s]
net.sf.j2s.ajax [trunk/sources/net.sf.j2s.ajax]
net.sf.j2s.core [branches/net.sf.j2s.core_3.6]
net.sf.j2s.html [trunk/sources/net.sf.j2s.html]
net.sf.j2s.java.core [trunk/sources/net.sf.j2s.java.core]
net.sf.j2s.java.junit [trunk/sources/net.sf.j2s.java.junit]
net.sf.j2s.java.org.eclipse.core.commands [trunk/sources/net.sf.j2s.java.org.eclipse.core.commands]
net.sf.j2s.java.org.eclipse.core.runtime [trunk/sources/net.sf.j2s.java.org.eclipse.core.runtime]
net.sf.j2s.java.org.eclipse.jface [trunk/sources/net.sf.j2s.java.org.eclipse.jface]
net.sf.j2s.java.org.eclipse.osgi [trunk/sources/net.sf.j2s.java.org.eclipse.osgi]
net.sf.j2s.java.org.eclipse.swt [trunk/sources/net.sf.j2s.java.org.eclipse.swt]
net.sf.j2s.lib [trunk/sources/net.sf.j2s.lib]
net.sf.j2s.ui [branches/net.sf.j2s.ui_3.6]
net.sf.j2s.update.site [trunk/sources/net.sf.j2s.update.site]

Figure 11.7: Import sources from SVN repository

Note
For commiting your changes back to java2script sources you need to be a member of the developer team. Just ask in Java2Script user group. Otherway, if you only want to checkout the sources, you don't need to use `https://` just `http://`.

11.2 Source organization

the Java2Script eclipse plugin is cosntituted by several projects working together. Some of these projects contains the eclipse plugin implementation, other contains the java implementation of required java APIs like java.lang, java.util, org.ecipse.swt, etc and other are only auxiliary. In the following table we list the projects that are part of Java2Script and the responsbility of each of them.

11.2.1 net.sf.j2s.core

It is an eclipse plugin that contains the java to javascript compiler. It is based on eclipse JDT for parsing the java code and translating it to javascript code.

11.2.2 net.sf.j2s.java.core

This project contains the java language and java API emulation for javascript. First of all it contains the standar java libraries implementation (java.lang, java.utils, etc) that are currently translated to javascript and used by your Java2Script application. At this time, a version of Apache Harmony is used. Worth mentioning that some core classes in java.lang like String, Class, Boolean are only implemented in javascript (there is no /java/lang/Boolean.java but a /java/lang/Boolean.js file). Also notice that files /java/lang/Class.js, /java/lang/CLassExt.js and /java/lang/ClassLoader.js actually contains the java language emulation for javascript, like the Object oriented programming emulation, java class loading emulation, etc.

11.2.3 net.sf.j2s.java.org.eclipse.swt

This project contains an implementation of the popular GUI toolkit SWT, the Eclipse Standar Widget Toolkit. This project will let you write your Rich Internet applications with SWT.

11.2.4 net.sf.j2s.lib

This is an auxiliary project in which is deposited the more like an auxiliary project into wich all the java libraries projects are comiler

11.2.5

11.3 Building the sources

There is a couple of things that should be done before building the sources. You only have this two things only once

The first is to decompress the file `net.sf.j2s.lib/j2slib.zip`. The easy way of doing so is to open `net.sf.j2s.lib/build/build.xml` in Ant view and run the `j2s.unzip` ant task

Second, the project `net.sf.j2s.ajax` requires JAVAX_SERVLET classpath variable to point to a .jar with the Java servlet specification. This is easy to infer looking at Eclipse `Problems` view.

You can create such a classpath variable pointing to Tomcat 5.0+'s servlet-apis.jar or plugin org.eclipse.tomcat's servlets.jar or other similar jar. Fortunately this is easy, first of all, locate the `servlet-api.jar` file in your computer. Probably you can find it on your Eclipse `plugins` directory (in my case it is located at `eclipse/plugins/javax.servlet_2.5.0-.v200910301333.jar`). Also, this .jar file comes in any Java web server, like Tomcat (in the case of Tomcat 5.5 it is at `common/lib/servlet-api.jar`)

Now that you have located `servlet-api.jar` file in your computer, go to Eclipse `Package Explorer` view, right click on the project `net.sf.j2s.ajax -> Build Path -> Configure Build Path...`. Go to the "Libraries" tab, double click the `JAVAX_SERVLET` variable, then click on `Variable...` button and the click on `New...` button. In the "New Variable Entry" dialog, enter `JAVAX_SERVLET` as the variable name and the path to servlet-api.jar as the variable value:

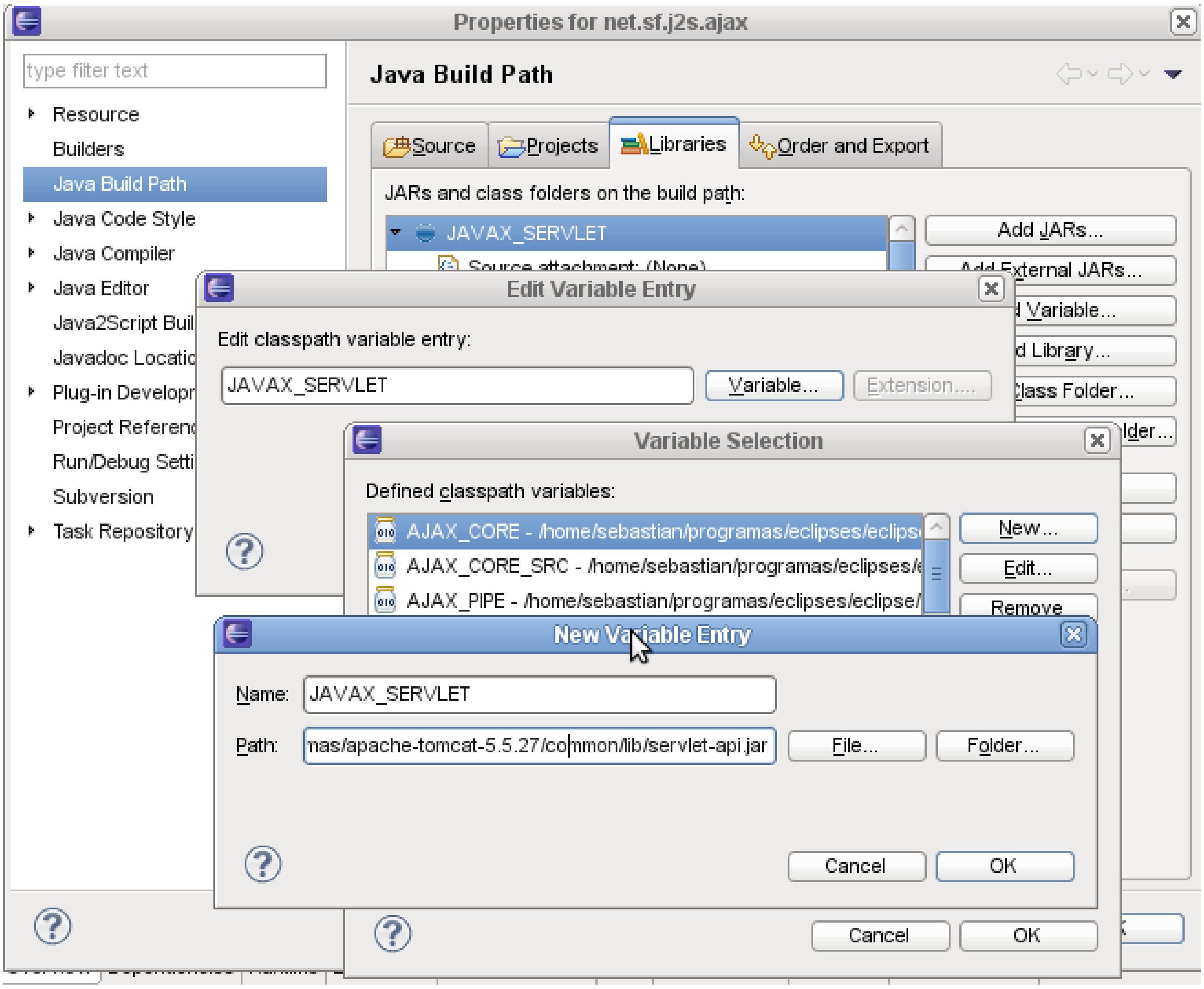


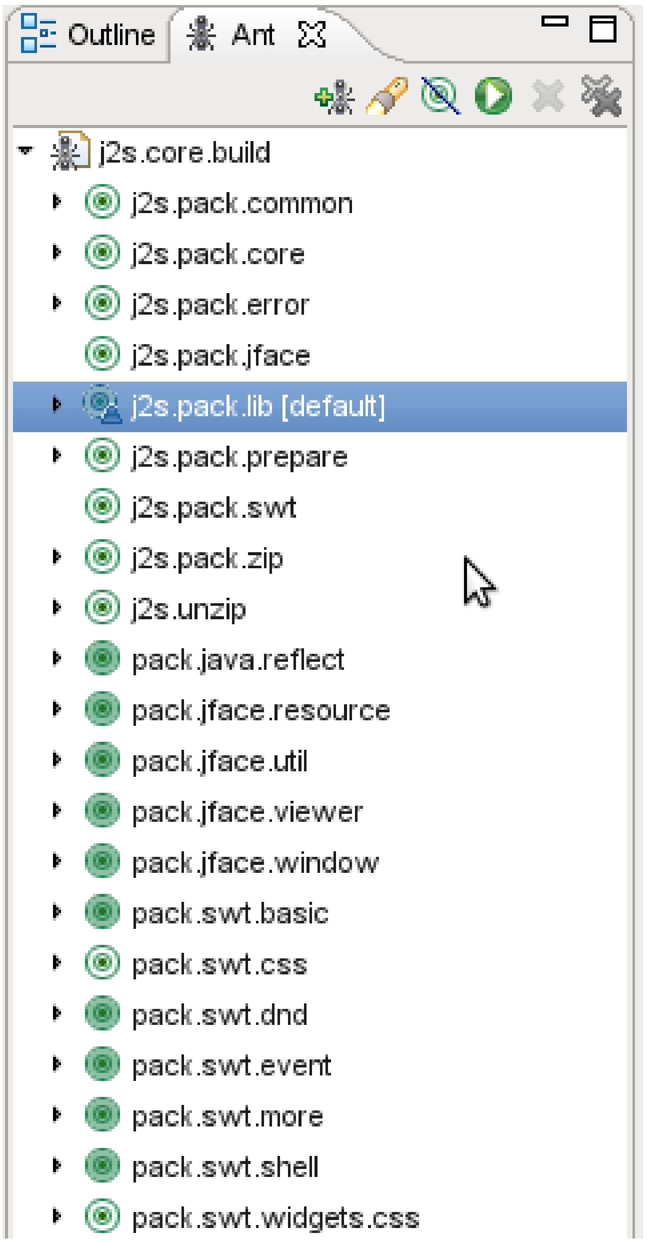
Figure 11.8: Running the Java2Script Eclipse plugin

After this there should be no errors in the "Problems" Eclipse view and all projects should be rebuilt. The first time, this may take a while. Remember that there are two compilers compiling some large projects like java.lang, java.util package, swt, and other java toolkits both to .class and to javascript.

As said in the previous section, Java2Script plugin contains two different kind of java sources. In one side we have the J2S eclipse plugin itself in projects net.sf.j2s.core and net.sf.j2s.ui and in the other side we have projects like net.sf.j2s.java.core and net.sf.j2s.java.org.eclipse.swt that are actually java libraries to be compiled into javascript and used by our Java2Script applications developed with the Java2Script plugin.

Both kind of projects will be built automatically [3] so we only have to save our files to impact the changes. Nevertheless, if we are modifying files that will be compiled to JavaScript, like the ones in project `net.sf.j2s.java.core`, we should also rebuild j2slib. It is easy. Just open `net.sf.j2s.lib/build/build.xml` in Ant view and run the `j2s.pack.lib` ant task:

[3]Only make sure that menu `Project->Build Automatically` is checked.



j2s.core.build
- j2s.pack.common
- j2s.pack.core
- j2s.pack.error
- j2s.pack.jface
- j2s.pack.lib [default]
- j2s.pack.prepare
- j2s.pack.swt
- j2s.pack.zip
- j2s.unzip
- pack.java.reflect
- pack.jface.resource
- pack.jface.util
- pack.jface.viewer
- pack.jface.window
- pack.swt.basic
- pack.swt.css
- pack.swt.dnd
- pack.swt.event
- pack.swt.more
- pack.swt.shell
- pack.swt.widgets.css

Figure 11.9: Packing j2slib

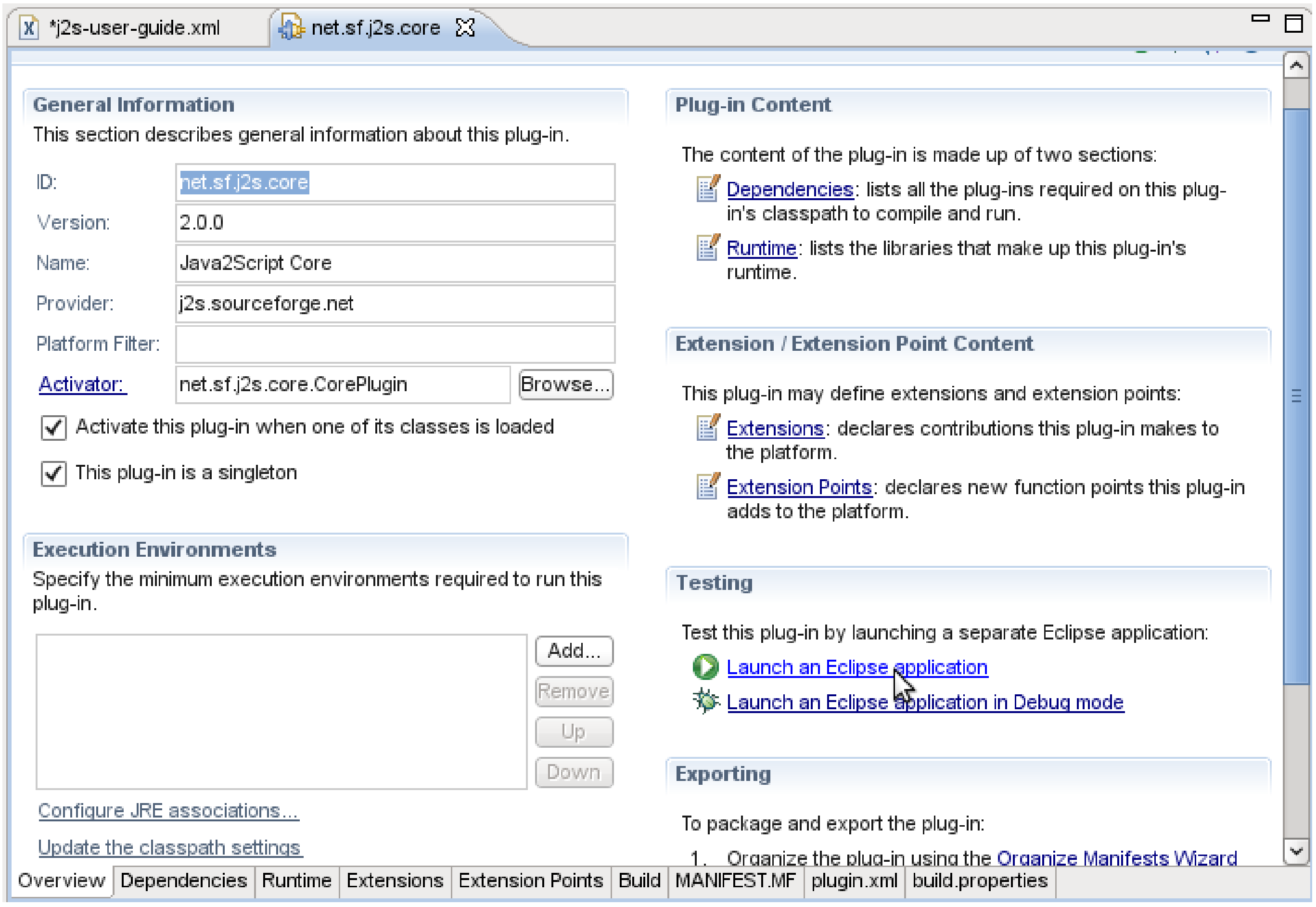
Important
Please remember to always run `ant j2s.pack.lib` when modifying Java files to be compiled to JavaScript before launching the Eclipse plugin like showed in next section.

Remember, as with any Java project, if you want to force project rebuild, you can always goto menu "Project -> Clean" and clean all projects: also be sure that "Project -> Build Automatically" is checked for automatically building when you change a file.

11.4 Running and debugging

Now that our sources are ready and builded, we want to run and debug the plugin. This is very easy, all you need is to open `net.sf.j2s.core/plugin.xml` and click the link ‘`Launch an Eclipse application`’ or the link ‘`Launch an`

```
Eclipse application in debug mode'
```


*j2s-user-guide.xml
net.sf.j2s.core

General Information
This section describes general information about this plug-in.

ID:	net.sf.j2s.core
Version:	2.0.0
Name:	Java2Script Core
Provider:	j2s.sourceforge.net
Platform Filter:	
Activator:	net.sf.j2s.core.CorePlugin

[✓] Activate this plug-in when one of its classes is loaded
[✓] This plug-in is a singleton

Execution Environments
Specify the minimum execution environments required to run this plug-in.

Add... Remove Up Down

Configure JRE associations...
Update the classpath settings

Plug-in Content
The content of the plug-in is made up of two sections:
Dependencies: lists all the plug-ins required on this plug-in's classpath to compile and run.
Runtime: lists the libraries that make up this plug-in's runtime.

Extension / Extension Point Content
This plug-in may define extensions and extension points:
Extensions: declares contributions this plug-in makes to the platform.
Extension Points: declares new function points this plug-in adds to the platform.

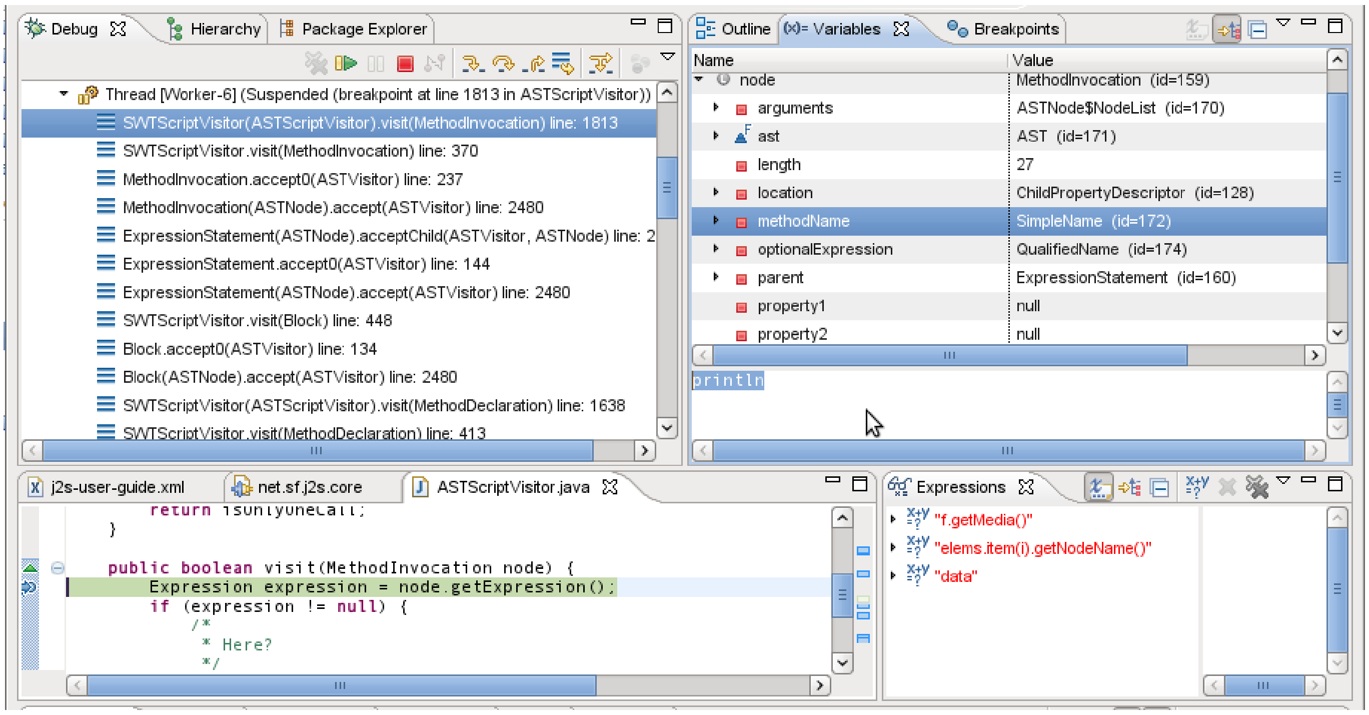
Testing
Test this plug-in by launching a separate Eclipse application:
Launch an Eclipse application
Launch an Eclipse application in Debug mode

Exporting
To package and export the plug-in:
1. Organize the plug-in using the Organize Manifests Wizard

Figure 11.10: Running the Java2Script Eclipse plugin

This will start a second Eclipse instance with our Java2Script plugin installed. Try to create a new Java2Script application and test your plugin's mdifications in that instance...

Like with any Eclipse plugin, rememeber that you can add breakpoints in Java code for better debbugging the Java to JavaScript compiler:



```
Thread [Worker-6] (Suspended (breakpoint at line 1813 in ASTScriptVisitor))
    SWTScriptVisitor(ASTScriptVisitor).visit(MethodInvocation) line: 1813
    SWTScriptVisitor.visit(MethodInvocation) line: 370
    MethodInvocation.accept0(ASTVisitor) line: 237
    MethodInvocation(ASTNode).accept(ASTVisitor) line: 2480
    ExpressionStatement(ASTNode).acceptChild(ASTVisitor, ASTNode) line: 2
    ExpressionStatement.accept0(ASTVisitor) line: 144
    ExpressionStatement(ASTNode).accept(ASTVisitor) line: 2480
    SWTScriptVisitor.visit(Block) line: 448
    Block.accept0(ASTVisitor) line: 134
    Block(ASTNode).accept(ASTVisitor) line: 2480
    SWTScriptVisitor(ASTScriptVisitor).visit(MethodDeclaration) line: 1638
    SWTScriptVisitor.visit(MethodDeclaration) line: 413
```

Name	Value
node	MethodInvocation (id=159)
arguments	ASTNode$NodeList (id=170)
ast	AST (id=171)
length	27
location	ChildPropertyDescriptor (id=128)
methodName	SimpleName (id=172)
optionalExpression	QualifiedName (id=174)
parent	ExpressionStatement (id=160)
property1	null
property2	null

```
println
```

```
        return isOnlyOneCall;
    }

    public boolean visit(MethodInvocation node) {
        Expression expression = node.getExpression();
        if (expression != null) {
            /*
             * Here?
             */
```

```
"f.getMedia()"
"elems.item(i).getNodeName()"
"data"
```

Figure 11.11: A breakpoint in some Java to JavaScript compiler class:

11.5 Exporting the plugin

So you make your changes and now you want to redistribute the plugin. As with any other Eclipse plugin, you have to export it as an eclipse plugin. It is very easy...

TODO. this is documented at : http://j2s.sourceforge.net/setup-j2s-from-subversion.html. this is what it sais:

Export the above 4 projects as "Deployable plug-ins and fragments". For example, you may select C:\eclipse as the directory of export destination. Project "net.sf.j2s.ajax" and "net.sf.j2s.lib" must be exported with "Package plug-ins as individual Jar archives" UNCHECKED. And project "net.sf.j2s.ajax" should be exported with "Include source code" option CHECKED, so later you can bind sources much easily.

11.6 How to Extend the Java2Script compiler

Now that you know how to install and run Java2Script from the sources, and know how to export your own version of Java2Script, we will describe a mechanism supported by Java2Script eclipse plugin that will let us extend the compiler easily, without modifying the compiler sources and without having to learn all compiler related classes internals for doing it.

Note
Original Java2Script articles about this topic are are the following:
http://j2s.sourceforge.net/articles/tutorial-extended-compiler.html
http://blog.java2script.org/2006/10/31/extending-java2script-compiler/

In this section you will learn how to customize how to customize the output of j2s compiler. You will be able to make an eclipse plugin that contribute to Java2Script Compiler with custom code for performing the translation of desired java language elements, like methods, classes, fields, expressions, statements, comments, etc to javascript. The main benefit of using an independent eclipse plugin is that you can isolate your customization code from java2script compiler nd so the same java2script distribution can be used for different java to javascript compiling scenarios.

Basically, this eclipse plugin must indicate what it want to customize, and provide the actual code translation for that parts. For example,

- analize each method invocation and add some javascript code before or after the invocation.
- analize each javadoc comment and search for a certain string or javadoc annotation inside it and add some javascript code before or after the java element documented with that javadoc
- add javascript code before or after a class declaration for classes that apply some conditions
- customize how java classes must be loaded in certain circunstances

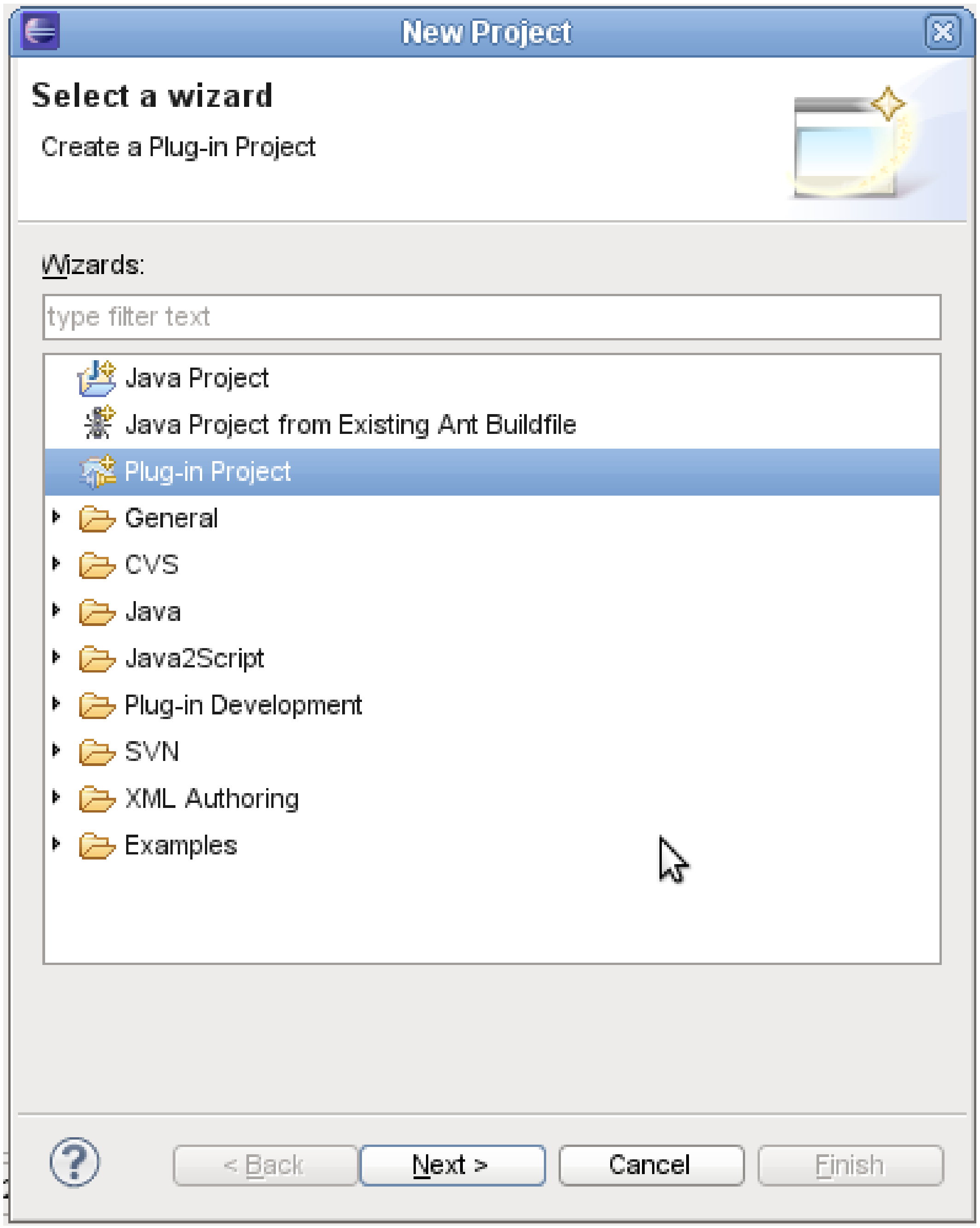
For developing the eclipse plugin we will use the eclipse plugin environment (PDE) that comes with common eclipse distributions. It is recommended but not mandatory to be familiarized with PDE, so you may want to read Plug-in Development Environment Guide .

We will also will need to model java elements like classes, methods, attributes, expressions, statements. For this, we will use eclipse Java Develpment Tools (JDT) and so, for taking full advantadge of this, it is recommended to know the basics about JDT. So, you may want to read JDT programmers guide

11.6.1 Creating a Java2Script compiler extension eclipse plugin

In this section we will create an eclipse plugin that contributes to Java2Script compiler with a very simple behavior: it will add a javascript comment before each method invocation, and then the normal method invocation javascript code.

so select File->New->Project...->Plugin Project



Select a wizard

Create a Plug-in Project

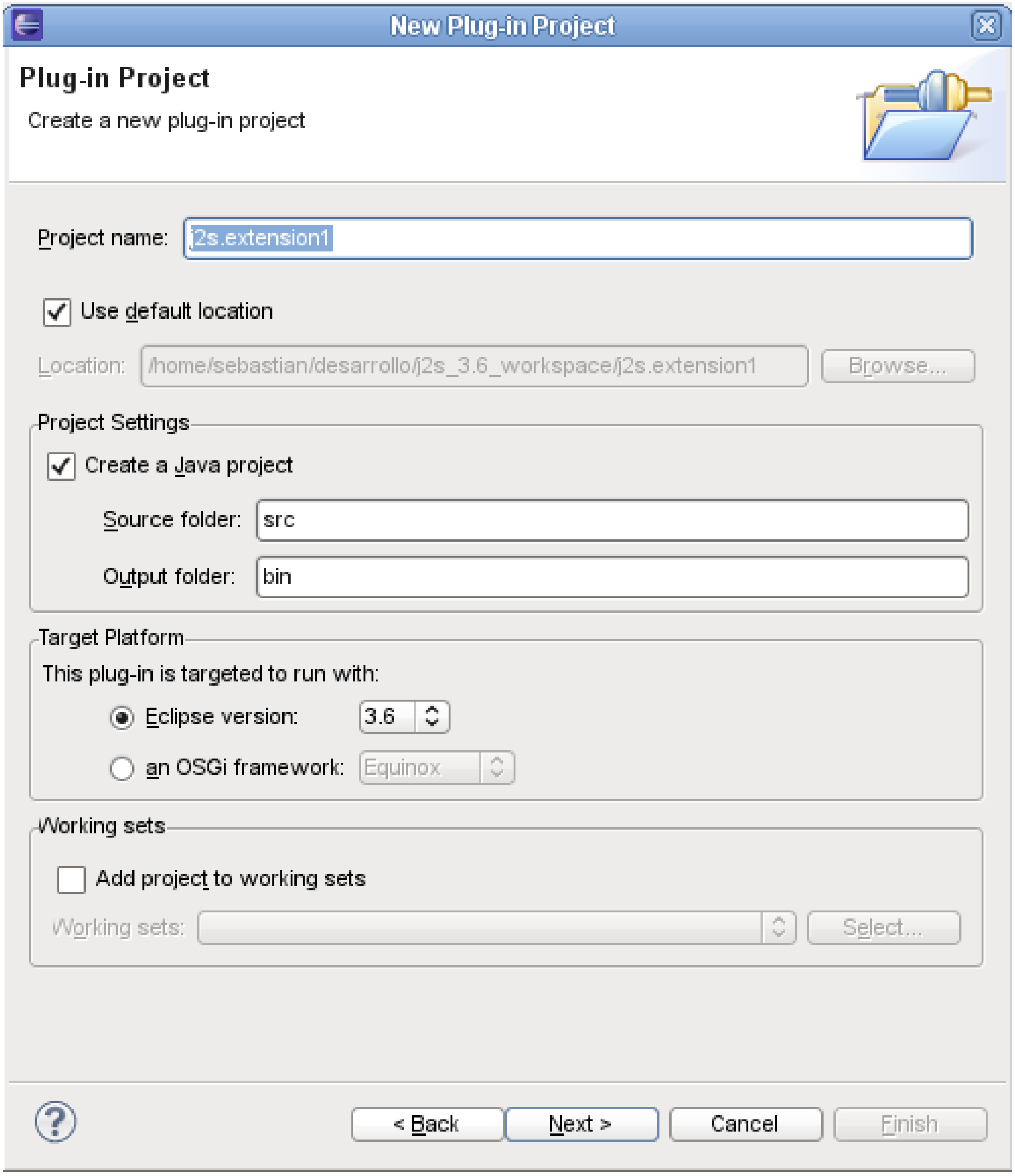
Wizards:

type filter text

- Java Project
- Java Project from Existing Ant Buildfile
- Plug-in Project
- General
- CVS
- Java
- Java2Script
- Plug-in Development
- SVN
- XML Authoring
- Examples

Figure 11.12: Creating a new plugin eclipse project

Press "next" button.



New Plug-in Project

Plug-in Project

Create a new plug-in project

Project name: j2s.extension1

[✓] Use default location

Location: /home/sebastian/desarrollo/j2s_3.6_workspace/j2s.extension1

Project Settings

[✓] Create a Java project

Source folder: src

Output folder: bin

Target Platform

This plug-in is targeted to run with:

● Eclipse version: 3.6

○ an OSGi framework: Equinox

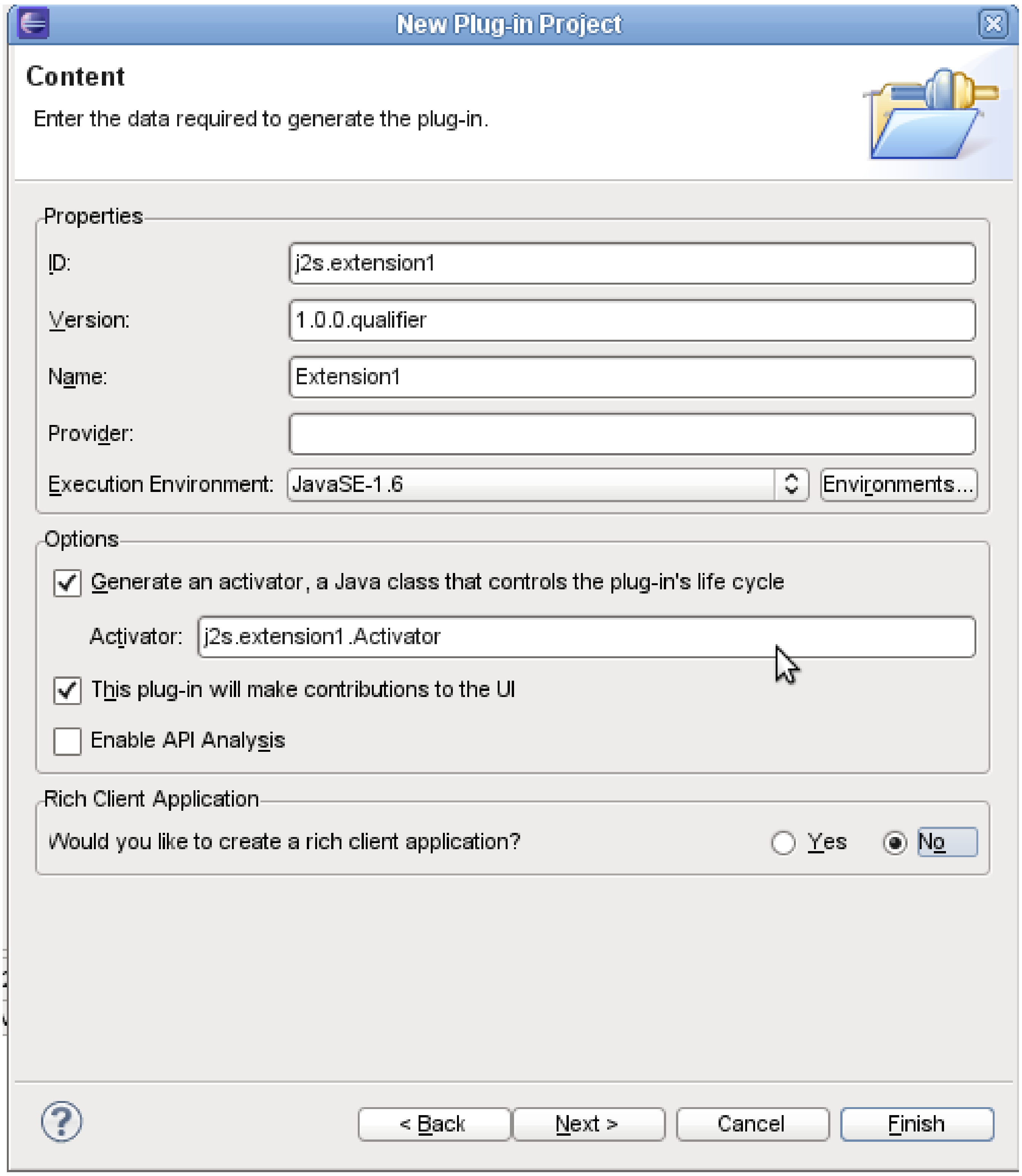
Working sets

[ ] Add project to working sets

Working sets:

Figure 11.13: Giving a name to the plugin

Press "next" button and in the following step make sure to choose "No" to "Would you like to create a rich client application?":



Content

Enter the data required to generate the plug-in.

Properties

ID: j2s.extension1

Version: 1.0.0.qualifier

Name: Extension1

Provider:

Execution Environment: JavaSE-1.6 Environments...

Options

[✓] Generate an activator, a Java class that controls the plug-in's life cycle

Activator: j2s.extension1.Activator

[✓] This plug-in will make contributions to the UI

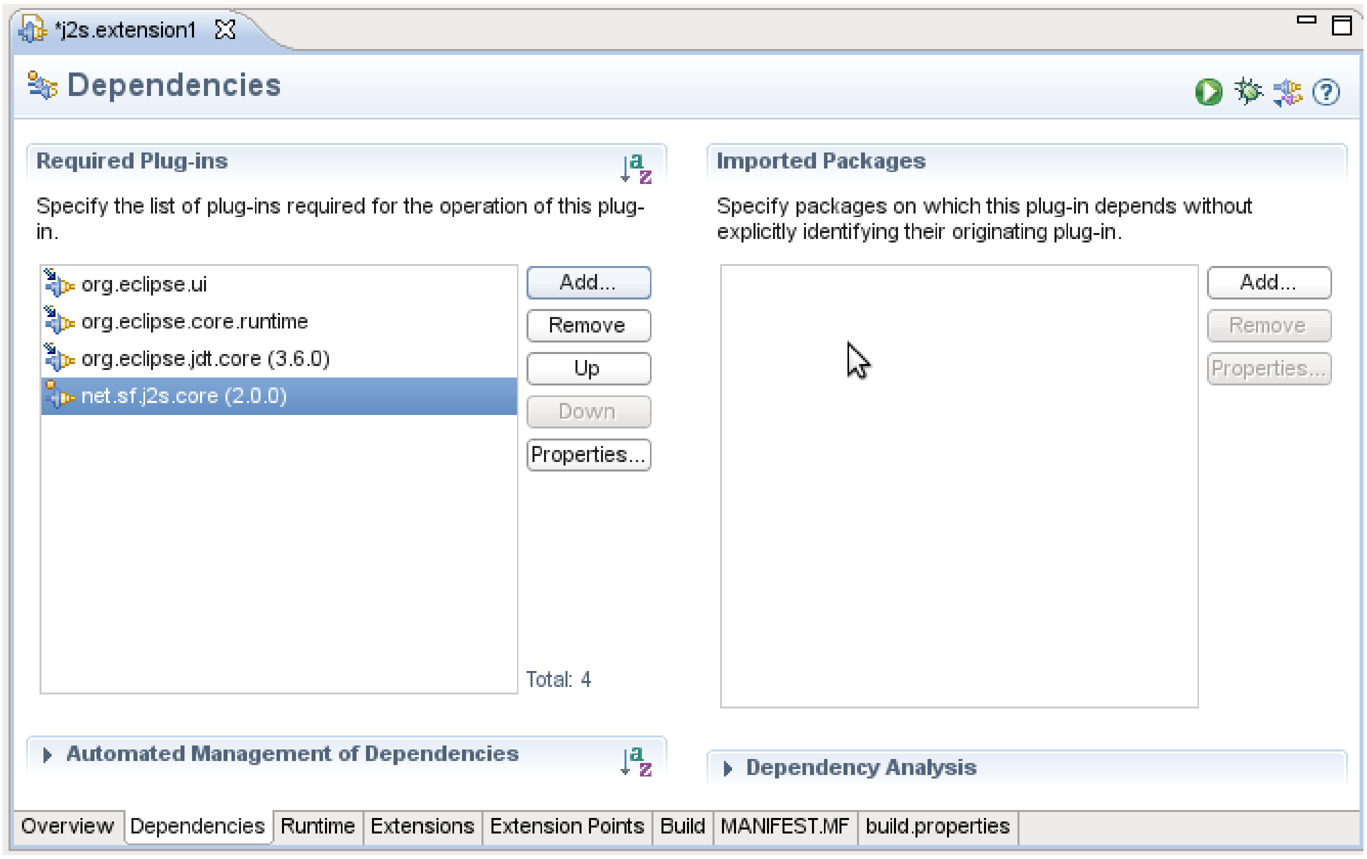
[ ] Enable API Analysis

Rich Client Application

Would you like to create a rich client application? Yes [✓ No]

Figure 11.14: new eclipse plugin project last step

Press finnish for creating your new J2S compiler extension plugin. Now we want to add "org.eclipse.jdt.core" and "net.sf.j2s.core" dependencies to out new plugin. For this, double click on file META-INF/MANIFEST.MF will open the eclipse plugin editor. Goto the "Dependencies" tab and add "org.eclipse.jdt.core" and "net.sf.j2s.core" as the dependencies:


*j2s.extension1

Dependencies

Required Plug-ins

Specify the list of plug-ins required for the operation of this plug-in.

- org.eclipse.ui
- org.eclipse.core.runtime
- org.eclipse.jdt.core (3.6.0)
- net.sf.j2s.core (2.0.0)

Total: 4

Imported Packages

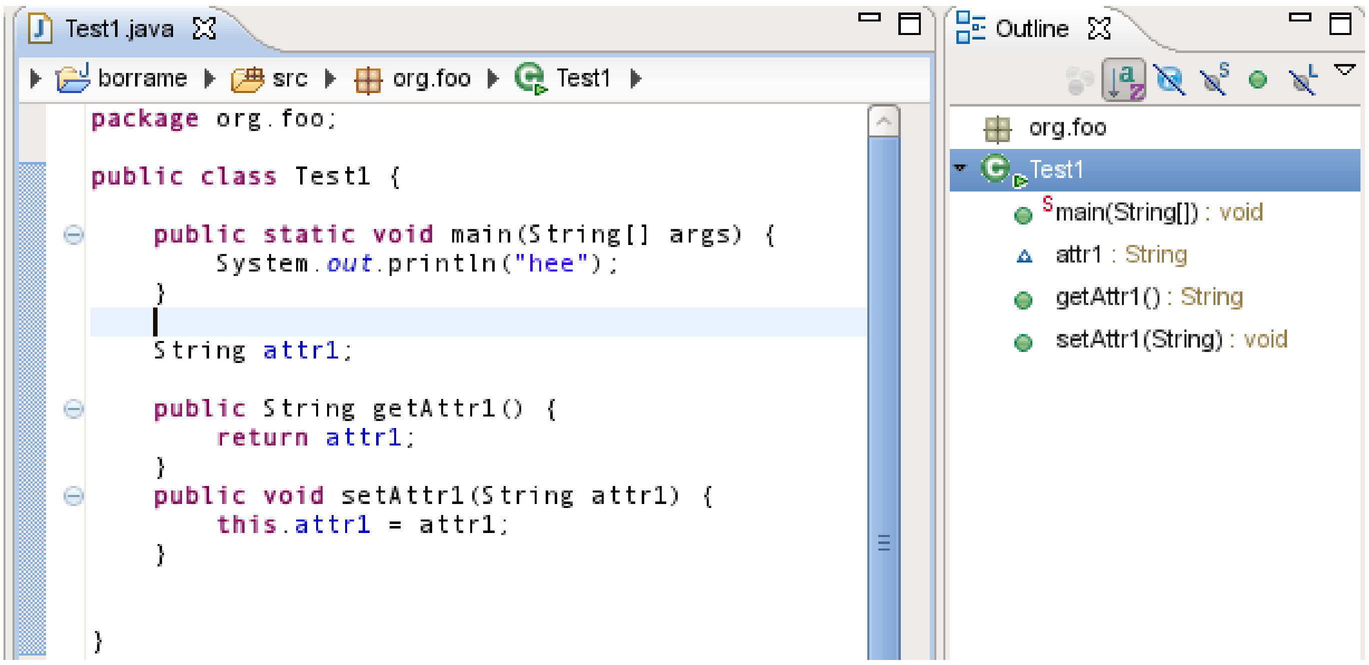
Specify packages on which this plug-in depends without explicitly identifying their originating plug-in.

Figure 11.15: eclipse plugin dependencies tab.

Note
At this point you have created an J2S extension plugin general project. All previous steps must be performed identically for all your J2s extension plugins.

Now we want to add an extension point from where our plugin will be plugged to the java2script compiler. Let's detail a little how Java2Script works and how our extenison is plugged into the java to javascript compiler.

First of all, the java2script compiler uses eclipse JDT java compiler for parsing all java code into a tree of *java elements* , called an Abstract syntax tree (AST). For example, a java compilation unit can contain a class declaration, with methods and fields declarations. A method contains a body, and the body some statements. Statements are composed by expresions, etc. In the following figure, the eclipse Outline view shows some of this java elements graphically:



```
package org.foo;

public class Test1 {

    public static void main(String[] args) {
        System.out.println("hee");
    }

    String attr1;

    public String getAttr1() {
        return attr1;
    }
    public void setAttr1(String attr1) {
        this.attr1 = attr1;
    }


}
```

```
org.foo
Test1
    main(String[]) : void
    attr1 : String
    getAttr1() : String
    setAttr1(String) : void
```

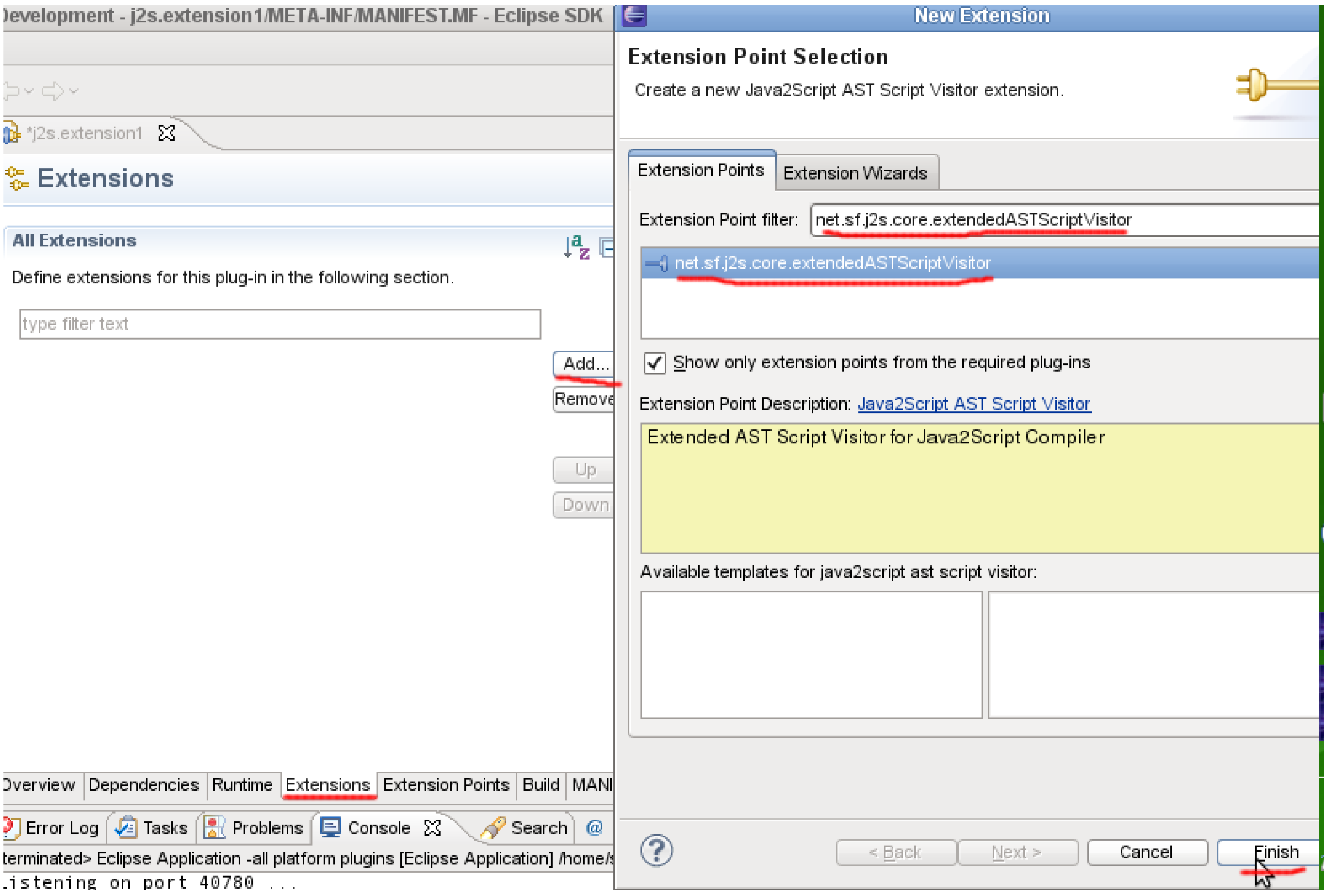
Figure 11.16: eclipse outline view showing some java elements of a compilation unit

Then the compiler uses "visitors" for converting java source codes to javascript. A visitor is an object which class contains overloaded "visit" methods for visiting each of java element types. Defining new visitors, you can overwrite the "visit" methods corresponding to the desired java element type that you want to customize its translation to javascript. Java2Script uses TWO kind of visitors for doing the total translation from java to javascript:

ASTScriptVisitor In one side we have an ASTScriptVisitor that is responsible for converting basic java elements to the corresponding javascript code. J2S provides with the default implementation net.sf.j2s.core.astvisitors.SWTScriptVisitor that can be extended for building customized AST script visitors.

DependencyASTVisitor The other type if visitor is DependencyASTVisitor that is responsible for creating a class dependency trees, in other words an import list for each java file. J2S provides with a default implemenation net.sf.j2s.core.astvisitors.SWTDependenc
It is recommended that the user extends this class for defining its own AST dependency visitors.

go to the "extensions" tab and add the extension point `net.sf.j2s.core.extendedASTScriptVisitor`:



Define extensions for this plug-in in the following section.

type filter text

```
terminated> Eclipse Application -all platform plugins [Eclipse Application] /home/
Listening on port 40780 ...
```

Extension Point Selection

Create a new Java2Script AST Script Visitor extension.

Extension Point filter: net.sf.j2s.core.extendedASTScriptVisitor

net.sf.j2s.core.extendedASTScriptVisitor

[✓] Show only extension points from the required plug-ins

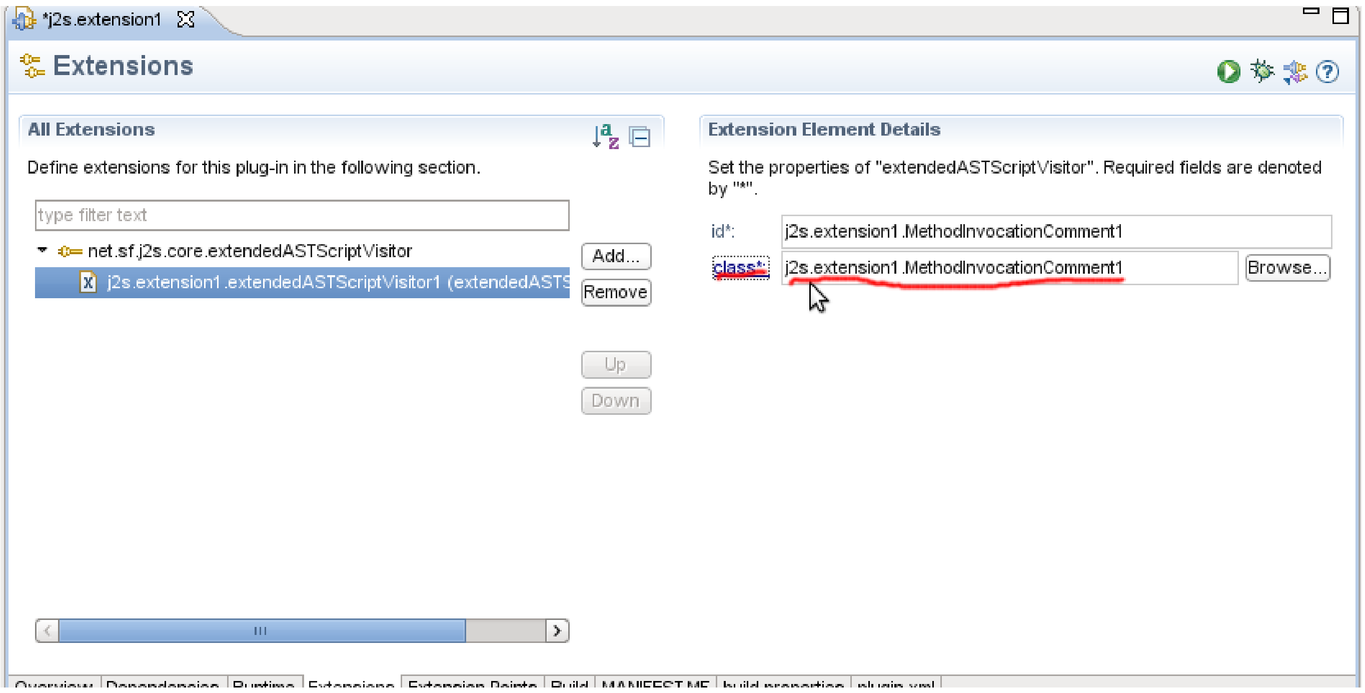
Extension Point Description: Java2Script AST Script Visitor

Extended AST Script Visitor for Java2Script Compiler

Available templates for java2script ast script visitor:

Figure 11.17: eclipse plugin - adding extension point

This extension point will allow us to provide with a class that will return the kind of visitor your extension want to provide. So choose the right id and class name. The id is very important because you must refence it in each Java2Script project that must be builded using this compiler extension, so remember it. In our name is "j2s.extension1.MethodInvocationComment1". Now click on "class" link for creating the new class:

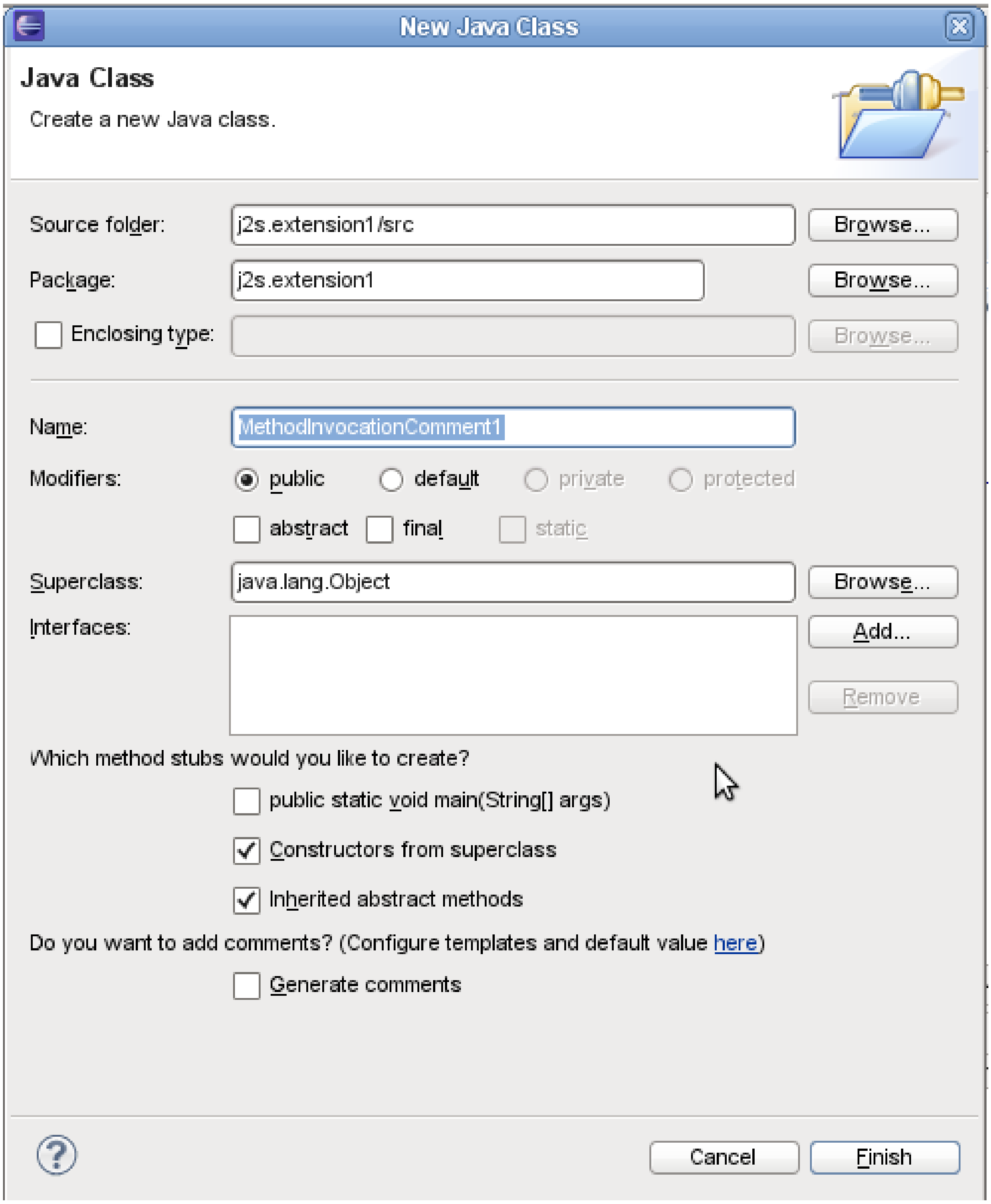

*j2s.extension1
Extensions
All Extensions
Define extensions for this plug-in in the following section.
type filter text
net.sf.j2s.core.extendedASTScriptVisitor
j2s.extension1.extendedASTScriptVisitor1 (extendedASTS
Add...
Remove
Up
Down
Extension Element Details
Browse...

Set the properties of "extendedASTScriptVisitor". Required fields are denoted by "*".

id*: j2s.extension1.MethodInvocationComment1

class*: j2s.extension1.MethodInvocationComment1

Figure 11.18: eclipse plugin - setting extension point name



New Java Class

Java Class

Create a new Java class.

Source folder: j2s.extension1/src

Package: j2s.extension1

Enclosing type: [ ]

Name: MethodInvocationComment1

Modifiers: [✓ public] default private protected

[ ] abstract [ ] final [ ] static

Superclass: java.lang.Object

Interfaces:

Which method stubs would you like to create?

[ ] public static void main(String[] args)

[✓] Constructors from superclass

[✓] Inherited abstract methods

Do you want to add comments? (Configure templates and default value here)

[ ] Generate comments

Figure 11.19: eclipse plugin - creating extension point class

Pressing finnish button for creating the class. This class msut implement interface net.sf.j2s.core.compiler.IExtendedVisitor. and should return the two visitors, ASTScriptVisitor and DependencyASTVisitor, to use by the Java2Script compiler for converting java to javascript.

In our example, we want only to provide a ScriptVisitor for changing how the java element "method invocation" is translated to javascript. We will use the default DependencyVisitor since we don't want customize anything of the java dependency tree:

```
package j2s.extension1;

import net.sf.j2s.core.astvisitors.ASTScriptVisitor;
import net.sf.j2s.core.astvisitors.DependencyASTVisitor;
import net.sf.j2s.core.astvisitors.SWTDependencyASTVisitor;
import net.sf.j2s.core.compiler.IExtendedVisitor;

public class MethodInvocationComment1 implements IExtendedVisitor {

  public MethodInvocationComment1() {
  }

  public ASTScriptVisitor getScriptVisitor() {❶
    return new MethodInvocationComment1ScriptVisitor();
  }

  public DependencyASTVisitor getDependencyVisitor() {❷
    return new DependencyASTVisitor();
  }
}
```

❶ we will use our own Script visitor, MethodInvocationComment1ScriptVisitor that we must create extending the default Script visitor net.sf.j2s.core.astvisitors.ASTScriptVisitor

❷ use the default J2S dependency visitor, net.sf.j2s.core.astvisitors.DependencyASTVisitor

Now, the only thing left is to create our custom Script Visitor that we named MethodInvocationComment1ScriptVisitor, so create the class:

```
package j2s.extension1;

import org.eclipse.jdt.core.dom.MethodInvocation;

import net.sf.j2s.core.astvisitors.SWTScriptVisitor;

public class MethodInvocationComment1ScriptVisitor extends SWTScriptVisitor {
  @Override
  public boolean visit(MethodInvocation node) {❶
    buffer.append("/* A COMMENT ADDED WITH our J2S compiler extension plugin */\n");❷
    return super.visit(node);❸
  }
}
```

❶ Overriding method visit(MethodInvocation node) we can customize how java method invocations are translated to javascript.

❷ the StringBuilder buffer field contain the actual javascript code. So we write our custom javascript comment before the method translation.

❸ invoking `super.visit(node)` will invoke the default implementation for java method invocation translation. We do it after writing our custom javascript code.

And that's it, your compiler extension is ready for use. You can export and install it as any other eclipse plugin. As usual, for exporting the extension, right click our project j2s.extension1 -> export... -> Plug-in development -> Deployable plug-ins and fragments. This will generate a plugins folder in the choosen destination directory with the extension plugin inside. You can redistribute it so other Java2Script users can install it in their systems like any other eclipse plugin.

Important
For using this extension in a Jav2Script project you must indicate the id of your J2S extension in your project's .j2s file. In our case you should add the following line to the .j2s file of the projects you want to compile with the extension:

```
j2s.compiler.visitor=j2s.extension1.MethodInvocationComment1
```

Now let's see how our J2S extension translate the following java class:

```
package j2s.extension1;

import org.eclipse.jdt.core.dom.MethodInvocation;

import net.sf.j2s.core.astvisitors.ASTScriptVisitor;

public class MethodInvocationComment1ScriptVisitor extends ASTScriptVisitor {
  @Override
  public boolean visit(MethodInvocation node) {
    buffer.append("/* A COMMENT ADDED WITH our J2S compiler extension plugin */\n");
    return super.visit(node);
  }
}
```

and the resulting javascript is

```
Clazz.declarePackage ("org.foo");
c$ = Clazz.declareType (org.foo, "Test1");
c$.main = Clazz.defineMethod (c$, "main",
function (args) {
/* A COMMENT ADDED WITH our J2S compiler extension plugin */
java.lang.System.out.println ("hee");
}, "~A");
```

As you can see, out javascript custom comment was appended just before the method declaration at `java.lang.System.-out.println ("hee");`

11.6.2 Other example: html attributes with javadoc annotations

In this second example we show how to load string from javadoc comments into java fields. In web applications, we usually have to reference html, json, xml or other web format strings in our code and sometimes it is nasty to do it in common java or javascript strings literals.

The following is the java class code that implements ASTScriptVisitor that will detect and load a string in javadoc after the tag `@j2sLoadString`.

```
package j2s.extension1;

import java.util.Iterator;
import java.util.List;

import net.sf.j2s.core.astvisitors.SWTScriptVisitor;

import org.eclipse.jdt.core.dom.FieldDeclaration;
import org.eclipse.jdt.core.dom.Javadoc;
import org.eclipse.jdt.core.dom.TagElement;
import org.eclipse.jdt.core.dom.VariableDeclarationFragment;

public class MethodInvocationComment1ScriptVisitor extends SWTScriptVisitor {

  @Override
  public boolean visit(FieldDeclaration node) {
    boolean ret = super.visit(node); /* first visit the default visitor implementation */

    Javadoc javadoc = node.getJavadoc();
    List tags = javadoc.tags();
    if(tags!=null&&tags.size()>0) {
      for (Iterator it = tags.iterator(); it.hasNext();) {
        Object o = (Object) it.next();
        if(o instanceof TagElement) {
          TagElement tag = (TagElement) o;
          String tagName = tag.getTagName();
          if(tagName!=null&&tagName.equals("@j2sLoadString")) { /* annotation found. all ↩
              the
            following javadoc comment until other tag is found will be loaded */
            StringBuffer sb = new StringBuffer();
            List frags = tag.fragments();
            for (Iterator it2 = frags.iterator(); it2.hasNext();) {
              sb.append(it2.next()+"\\n");
            }
            List fieldDeclFrags = node.fragments();
            /* now that we have the @j2sLoadString string, we add the javascript statement ↩
                this.attributeName="str.." for loading the string field.*/
            if(fieldDeclFrags!=null&&fieldDeclFrags.size()>0) {
              VariableDeclarationFragment f1 = (VariableDeclarationFragment) fieldDeclFrags ↩
                  .get(0);
              String str = sb.toString();
              buffer.append("this."+f1.getName()+"=\’");
              buffer.append(str);
              buffer.append("\’;\n");
            }
          }
        }
      }
    }
    return ret;
  }
}
```

Using this compiler extension, you will be able to code complex strings like json, html, xml, etc inside javadoc. The following java test program define two fields, htmlCode1 and jsonCode1 which content is loaded from javadoc and not from string literals. As you can see there is much easier to put other language code inside javadoc instead using java string, in which you have to quote and concatenate:

```
  package org.foo;

public class Test1 {

  public static void main(String[] args) {
    Test1 test1 = new Test1();
    System.out.println("loaded html code: "+test1.getHtmlCode1());
    System.out.println("loaded json code: "+test1.getJsonCode1());
    System.out.println("lot easier isn’t it?");
  }
  /** this field will be loaded with the fololowing string:
   * @j2sLoadString
   * <h1>this is an <b>html</b> string loaded from javadoc</h1>
   */
  String htmlCode1;

  /**
   * this other too
   * @j2s LoadString
   * {p1: [1, 2, 3], p2: {p21: "thi sis a json loaded from javadoc"}}
   */
  String jsonCode1;

  public String getHtmlCode1() {
    return htmlCode1;
  }

  public String getJsonCode1() {
    return jsonCode1;
  }
}
```

Compiling this using our new compiler extension, the output will be the following when executing this as a Java2Script application:

```
loaded html code:
<h1>this is an <b>html</b> string loaded from javadoc</h1>

loaded json code: {p1: [1, 2, 3], p2: {p21: "thi sis a json loaded from javadoc"}}

lot easier isn’t it?
```

I hope the reader can realize how helpfull this particular example can be, how easy it was to develope an independent Java2Script compiler extension that, in this particular case introduced a new @j2s directive for defining string class fields inside javadocs.

Appendix A

J2S Java Emulation API

Any sufficiently advanced technology is indistinguishable from magic.

—Arthur C. Clarke

In this section we will examine Java2Script support for java emulation. When you include the j2slib.z.js script in your html documents Java2Script provides 2 types of APIs:

- **Java Language API** . As we have seen in .js generated files, the object `Clazz` contains methods for that emulates the java language leting the javascript programmer to declare java elements like package, classes, methods and so on.
- **Java Runtime API** . As we have seen in the html files generated by Java2Script when we run a J2S application, the `Clazz-Loader` object contains an API for configuring the java runtime, for example, leting the javascript programmer configure the "java classpath", loading a set of classes and executing some java code when those classes are available.

In this chapter we try to document the relevant API functions of this two javascript objects.

A.1 Java Language Emulation API

When including the script j2slib.z.js, there will be available the object Clazz that contains functions for "doing java" in javascript. We have already examine this kind of JavaScript code generated by J2S in Section 8.1.

a lot of documentation is at http://j2s.sourceforge.net/articles/oop-in-js-by-j2s.html. TODO: include that in this section???

Clazz.defineMethod = function (clazzThis, funName, funBody, funParams)

define a java method. the doc says:

```
  /*
 * Define method for the class with the given method name and method
 * body and method parameter signature.
 *
 * @param clazzThis host class in which the method to be defined
 * @param funName method name
 * @param funBody function object, e.g function () { ... }
 * @param funParams paramether signature, e.g ["string", "number"]
 * @return method of the given name. The method may be funBody or a wrapper
 * of the given funBody.
 */
/* public */
Clazz.defineMethod = function (clazzThis, funName, funBody, funParams)
```

```
example defining an instance method:
TODO

example defining a class method:
TODO
```

A.2 Java Runtime Emulation API

In Section 8.2 we have examined how a java program is executed in an html document using the object ClazzLoader provided by J2S that contains method for configuring the java runtime with javascript, like setting the java classpath, loading classes and invoking java code. Here we will document all relevant functions of that object.

ClazzLoader.packageClasspath (pkg, base, index)

TODO: the index arguments, if true, a file .package will try to be parsed. this file can be used by j2s library autors to configure how a j2s library must be loaded.

ClazzLoader.setPrimaryFolder = function(bin)

TODO

ClazzLoader.ignore = function ()

Makes the J2s compiler to ignore (not load) an array of classes. accepts an array of classnames to ignore. Used when we put set some classes as "Abandoned" in the project Java2Script properties page.

ClazzLoader.jarClasspath

Makes the J2s compiler to ignore (not load) an array of classes. accepts an array of classnames to ignore. Used when we put set some classes as "Abandoned" in the project Java2Script properties page. ClazzLoader.jarClasspath (base + "util/Collections.js", ["java.util.Collections", "java.util.Collections.CheckedCollection", "java.util.Collections.CheckedList", "java.util.Collections.CheckedLi "java.util.Collections.CheckedMap", ...]);

Appendix B

GNU Free Documentation License

Version 1.3, 3 November 2008

Copyright © 2000, 2001, 2002, 2007, 2008 Free Software Foundation, Inc.

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

2. VERBATIM COPYING

3. COPYING IN QUANTITY

copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled “Endorsements”, provided it contains nothing but endorsements of your Modified Version by various parties — for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled “History” in the various original documents, forming one section Entitled “History”; likewise combine any sections Entitled “Acknowledgements”, and any sections Entitled “Dedications”. You must delete all sections Entitled “Endorsements”.

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an “aggregate” if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

Appendix C

Index