Using Third-Party Libraries with Eclipse and the BDT

By Nick Decker

One of the benefits afforded to BBj developers is that they can mix standard Business BASIC syntax with Java syntax. Even better, BBj gives developers the ability to add third-party JAR files to their BBjServices classpath, opening up new worlds of potential. BBj developers can easily take advantage of hundreds of Java libraries that cover the gamut from utilitarian, such as standardized logging, to reading, writing, and parsing a variety of data formats including JSON, XML, Comma Separated Value, and HTML. This article takes a look at adding jsoup, a free and commonly-used Java HTML parser library. The jsoup library serves as a jack-of-all-trades, providing the ability to:

- Scrape and parse HTML from a URL, file, or string
- Find and extract specific data via the DOM (Document Object Model) or CSS selectors
- Manipulate HTML elements, attributes, and text
- Sanitize untrusted HTML to prevent XSS (Cross-Site Scripting) attacks
- Resolve missing closing tags, escape HTML entities, standardize indention, and output tidy (pretty printed) HTML.

Getting Started

Before we begin, ensure that you already have a BBj development environment set up using Eclipse and the Business BASIC Development Toolkit (BDT) plug-in. If you do not yet have Eclipse fully configured, you will want to visit the Eclipse Plug-ins page on the BASIS website. Ensuring that Eclipse is configured for basic BBj development is a necessary first step, but later in this article we will make a few more specific configuration changes to prepare for our jsoup project.

JAR Files and Classpaths

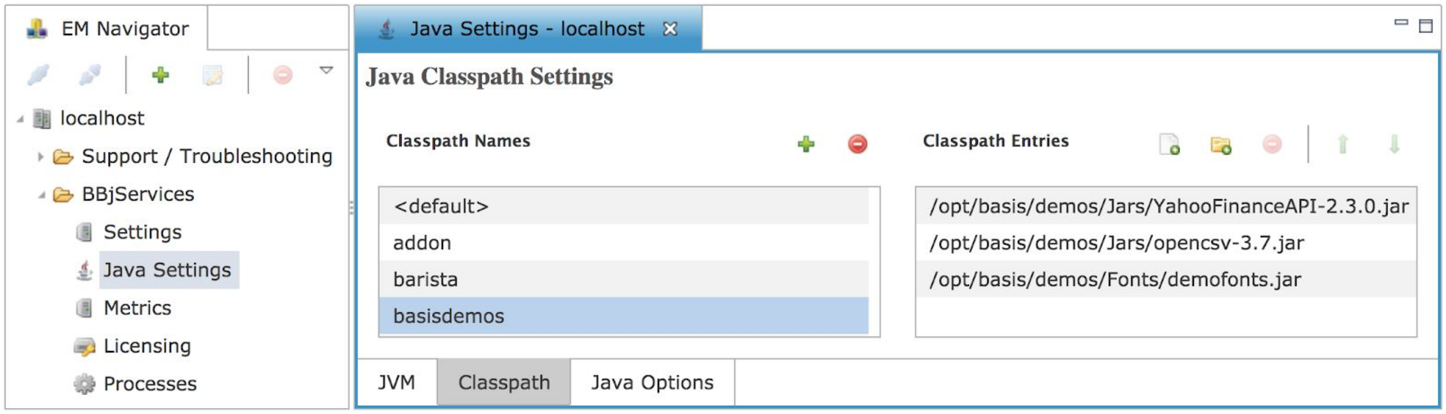
In order for a BBj program to access the Java code in a third-party library, we must first tell BBjServices about the JAR file. We can do this by adding the JAR file to BBjService's default classpath or via a Session-Specific Classpath (SSCP). After you add a JAR to the default classpath, all BBj interpreters will be able to access the library. If you want to limit which programs can access the library, you can create an SSCP that contains that JAR file. For an overview of SSCPs and use-cases, see the Session-specific Classpath documentation. The Calling Custom Java from BBj Advantage article covers this concept in greater detail and goes several steps further by describing how to make your own custom Java classes available to your BBj programs.

Downloading the jsoup JAR File

For this example, we will create a new SSCP to segregate the library and keep the default BBj classpath pristine. First, create a **`jars`** subdirectory underneath the **`<BBjHome>`** directory. Next, download the latest jsoup core JAR file from its download page. At the time of this publication, the latest released versions that we will be using are BBj 19.00 and jsoup 1.11.3. Finally, move the downloaded jsoup JAR file into the newly-created **`jars`** subdirectory so that it is easier to locate and replace with future updates.

Creating a New SSCP

Now that our jsoup JAR file is easily accessible, we will launch Enterprise Manager to create our SSCP. After authenticating, navigate to the **BBjServices** > **Java Settings** section and select the bottom **Classpath** tab. The resultant **Java Classpath Settings** panel shown in **Figure 1** has two lists side-by-side with the left one displaying the defined classpaths and the right one displaying the JAR files included in the selected SSCP.



Classpath Names	Classpath Entries
<default>	/opt/basis/demos/Jars/YahooFinanceAPI-2.3.0.jar
addon	/opt/basis/demos/Jars/opencsv-3.7.jar
barista	/opt/basis/demos/Fonts/demofonts.jar
basisdemos	

Figure 1. Viewing the list of SSCPs in Enterprise Manager

Creating a new SSCP is as easy as clicking the [Add] toolbutton above the Classpath Entries list and then providing a name. We named ours *jsoup*, so now there is a new entry in the alphabetized SSCP list. The last step involves adding the actual JAR file to the new SSCP, which can be initiated by clicking the [Add a JAR] toolbutton. After adding the **`<BBjHome>/jars/jsoup-1.11.3.jar`** file and clicking the [Save] button, we have a working *jsoup* classpath as shown in **Figure 2**.

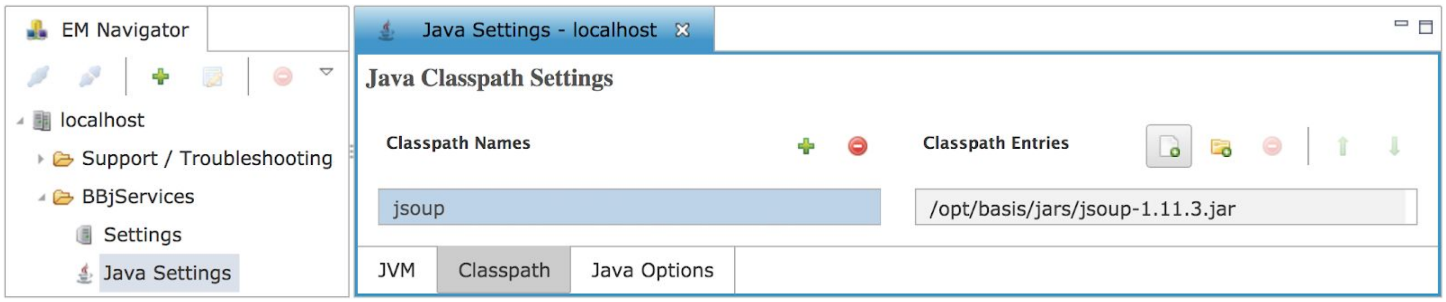


Figure 2. The newly-created *jsoup* SSCP and its classpath entry with the jsoup JAR

Preparing Our Development Environment in Eclipse

Now that we have a new *jsoup* classpath defined, we can finalize our Eclipse configuration for our new project. Because we just added a new classpath to BBjServices, we should restart Eclipse if it is already running to ensure that the BDT has an updated classpath that includes our *jsoup* entry. Once Eclipse is up and running again, we will create a new BBj Runtime Environment (BRE). To do this, go to the **BDT Explorer** tab in Eclipse, expand the **BBj Installation** disclosure triangle, then right-click the **BBj Runtime Environments** node. From the popup menu, select **Create New BRE**, as shown in **Figure 3**.

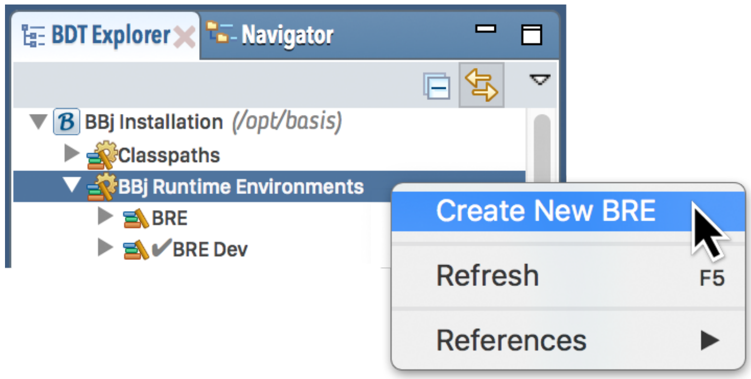
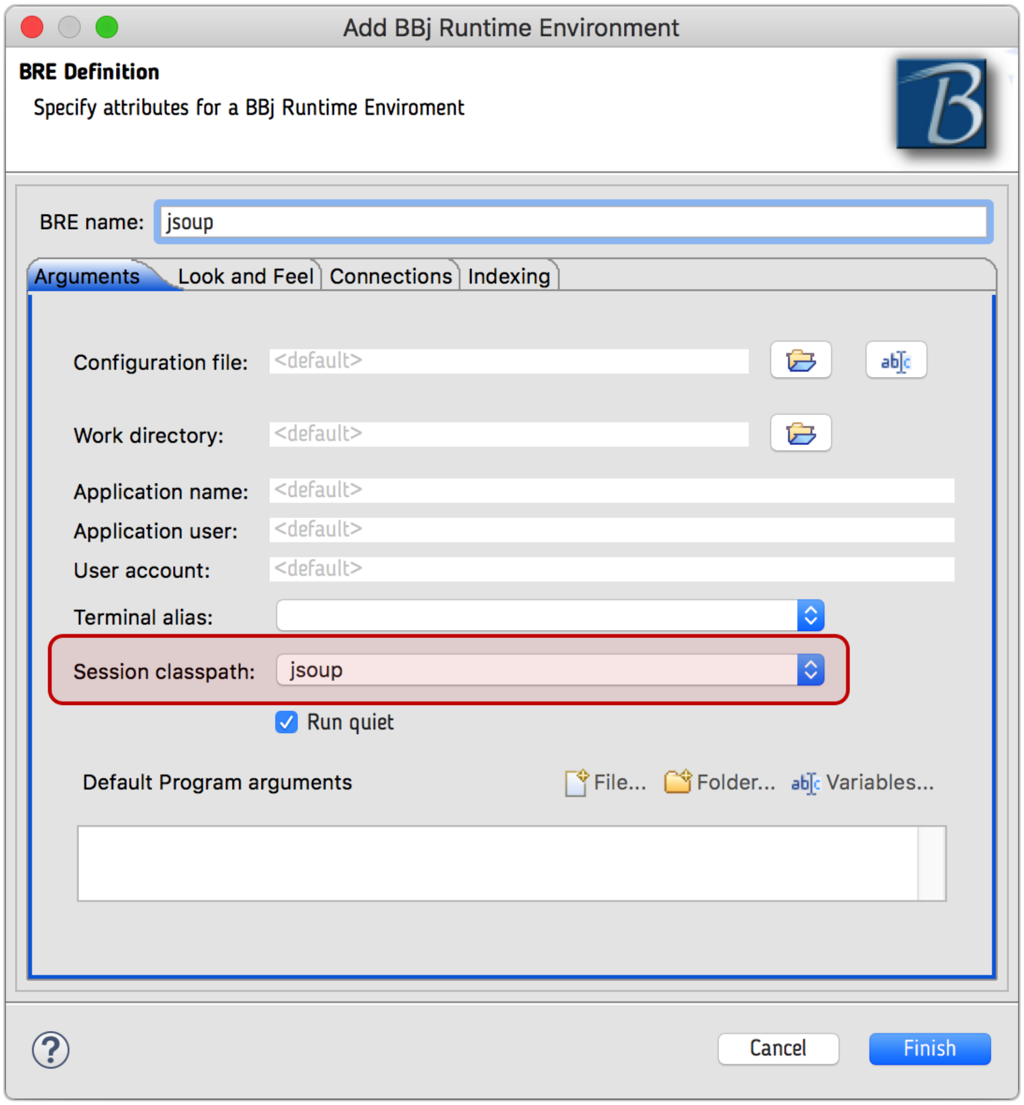


Figure 3. Creating a new BRE in Eclipse

Eclipse responds by displaying an **Add BBj Runtime Environment** window, which lets us define the new BRE. To keep everything well-organized and explicit, we will name the new BRE *jsoup*. The critical next step is to specify the *jsoup* classpath that we recently created in Enterprise Manager; we can leave all other parameters at their defaults. **Figure 4** shows the resultant window and highlights the selected *jsoup* classpath.


Add BBj Runtime Environment

BRE Definition

Specify attributes for a BBj Runtime Enviroment

BRE name: jsoup

Arguments | Look and Feel | Connections | Indexing

Configuration file: <default>

Work directory: <default>

Application name: <default>

Application user: <default>

User account: <default>

Terminal alias:

Session classpath: jsoup

[✓] Run quiet

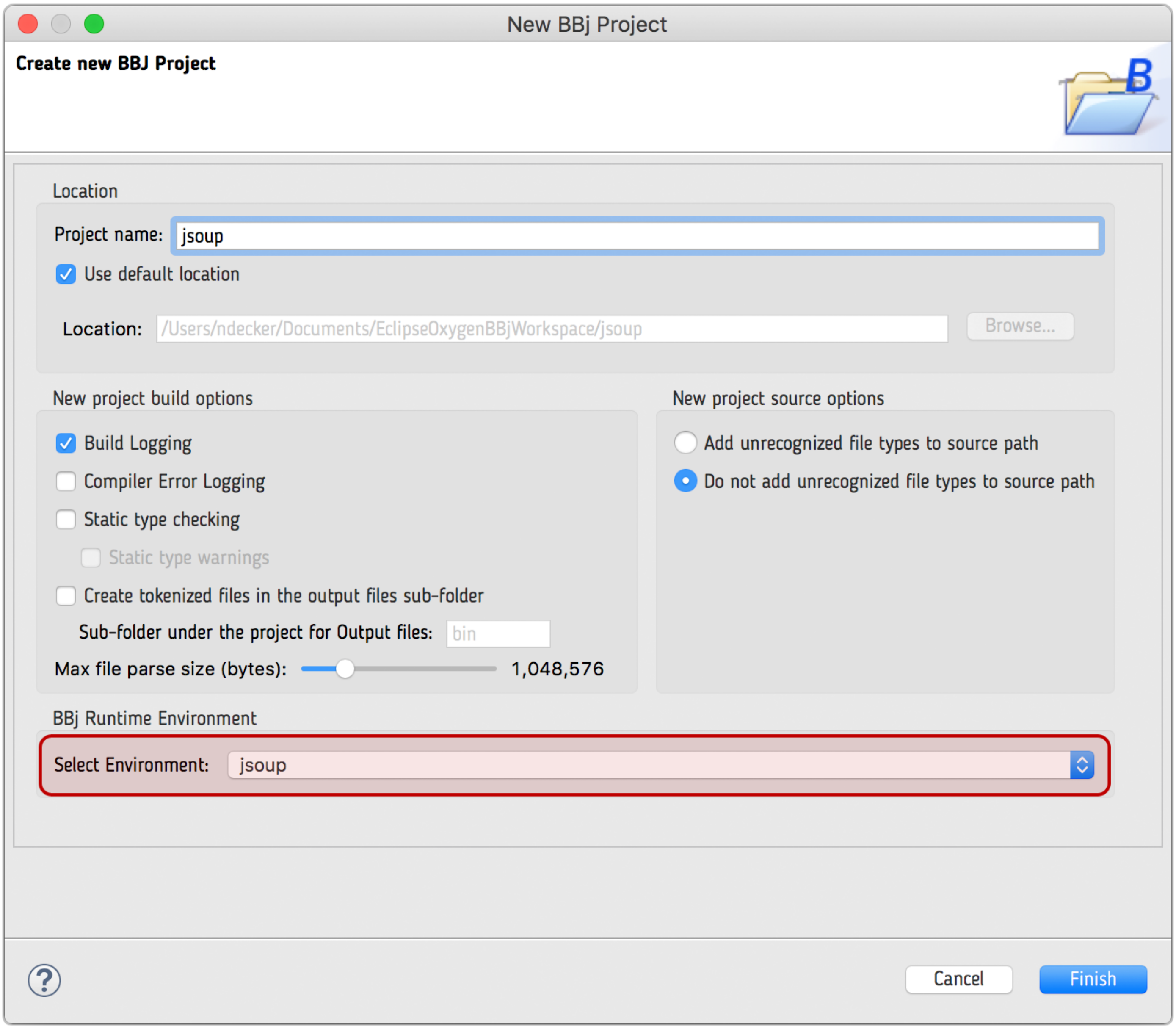
Default Program arguments

Figure 4. Creating a new *jsoup* BRE in Eclipse

Clicking the [Finish] button completes the new BRE definition. Eclipse may then display a window, notifying you that the projects' build states have changed and prompting you to rebuild existing projects. If that occurs, click the [Cancel] button to dismiss the window. At this point, we can expand the **Classpaths** and **BBj Runtime Environment** nodes shown earlier in **Figure 3**. If everything is in order, Eclipse displays a new *jsoup* entry under both nodes, and then we're ready to create a new BBj Project.

Creating a New Project

We are now ready to create a new project for our program. You can do this several ways in Eclipse, but the quickest is to select **File** > **New** > **BBj Project** from the Eclipse menu. Eclipse displays its New BBj Project window, which we can modify as shown in **Figure 5**.


New BBj Project

Create new BBJ Project

Location

Project name: jsoup

[✓] Use default location

Location: /Users/ndecker/Documents/EclipseOxygenBBjWorkspace/jsoup

New project build options

[✓] Build Logging

[ ] Compiler Error Logging

[ ] Static type checking

[ ] Static type warnings

[ ] Create tokenized files in the output files sub-folder

Sub-folder under the project for Output files: bin

Max file parse size (bytes): 1,048,576

New project source options

○ Add unrecognized file types to source path

● Do not add unrecognized file types to source path

BBj Runtime Environment

Select Environment: jsoup

Figure 5. Defining a new *jsoup* project in Eclipse

Note that we have named the project *jsoup* and have selected the *jsoup* BRE that we defined in **Figure 3**. We will leave the other options at their default settings, which means that Eclipse will create a **`jsoup`** folder under our chosen workspace directory to keep our project files. After clicking the [Finish] button to commit our changes, Eclipse creates and selects a new *jsoup* entry in the BDT Explorer, shown in **Figure 6**.

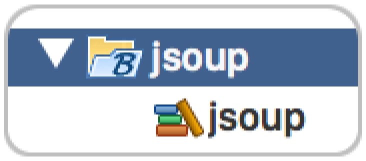


Figure 6. Our new jsoup project selected in the BDT Explorer

Our jsoup project is now completely defined and ready to go. The only entry in the project folder in the BDT Explorer is our jsoup BRE, but inspecting the **`jsoup`** folder from the operating system level reveals that it contains hidden BDT configuration files that define the project's settings.

Reviewing Our Progress

By following all the previous steps in the document, we have given the BDT enough information to know everything about the jsoup library. As a quick review, we have:

1. Downloaded and placed the `jsoup-1.11.3.jar` JAR into the `<BBjHome>/jars` directory.
2. Created an SSCP in Enterprise Manager called *jsoup* that references this JAR file.
3. Created an Eclipse BBj Runtime Environment that uses the *jsoup* SSCP.
4. Created an Eclipse project that uses the *jsoup* BRE.

This means that we should now be able to access code completion for the jsoup library in any new programs that we create in our *jsoup* project. Why? Because the BDT editor in which we write our BBj programs knows about the *jsoup* project, *jsoup* BRE, *jsoup* SSCP, and ultimately the `jsoup-1.11.3.jar` JAR file on our drive. Since the editor is now equipped with everything it needs to know about jsoup, it can examine the JAR file's contents and figure out all the classes and methods it has to offer. The jsoup JAR file contains tons of classes and methods, which we can see by executing the Java `jar` command from the `<BBjHome>/jars` directory. Note that this syntax requires that you have the JDK's `bin` directory in your path. Without that, you would need to provide the full path to the jar executable.

```
jar tf jsoup-1.11.3.jar
```

Having full access to the jsoup library classes and methods is critical for code completion, and the BDT will expedite our coding by capitalizing on that new-found information.

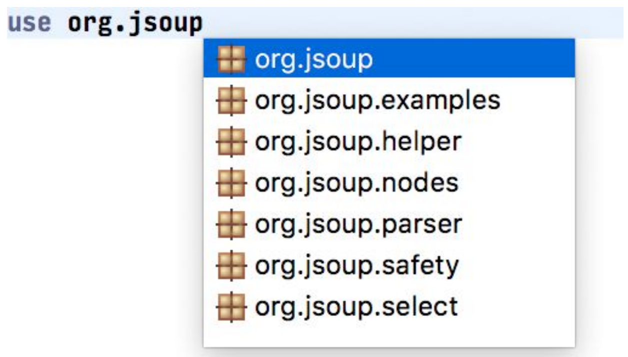
Creating a BBj Program

Now that all the JAR, SSCP, BRE, and Eclipse project settings are complete, it is time to start some actual programming. To begin, we create a new **BBj Source File** either via the Eclipse **File** > **New** menu option or by right-clicking on the *jsoup* project folder. In the spirit of keeping everything consistent, we will name our program file `jsoup.bbj` in the resultant **Create new BBj File** dialog.

Rather than being empty, our new program contains a default header courtesy of BDT's built-in BBj Source File Template. The template was selectable in the **Create new BBj File** dialog, and you can create your own templates if desired.

Testing Code Completion

Let's now put all our configuration efforts to the test. Hold your breath, and type in `use org.jsoup.` If Eclipse's code completion does not trigger automatically, you can invoke it manually by typing [Ctrl]+[Space]. Assuming that everything is set up correctly, you will be rewarded with a code completion popup as shown in **Figure 7**. You can now breathe a sigh of relief, as our configuration has been confirmed and the BDT is able to offer us content assistance for the jsoup library.



```
use org.jsoup
```

Figure 7. Activating code completion for the first time in a USE statement

If you have previously disabled content assist auto-activation, this can be turned on and set to a delay factor that best suits your programming style. You can find these settings in the **BDT** section of Eclipse's preferences under **Editor** > **Content Assist**. By default, the [Ctrl]+[Space] keystroke combination forces the BDT to invoke content assist. This may come in handy if you have made mistakes while typing and have gone back and forth a bit, only to find that the auto-activation has given up hope of triggering the completion list.

Writing Our First jsoup Program - *list all links in a given URL*

Our `use` statement from **Figure 7** activated code completion for the jsoup library, displaying several available classes that matched our typed text. At this point, though, we know very little about the jsoup library and even less about which classes we will eventually want to use. Instead of starting from scratch with the documentation, we will take a shortcut by visiting the jsoup cookbook page. Item 10 in the list is an example Java program that lists links in a given URL. That is a good place to start, because we will have the luxury of reading someone else's code that is proven to work correctly. All we have to do is figure out which portions of the example we want to replicate, then port that Java code to BBj.

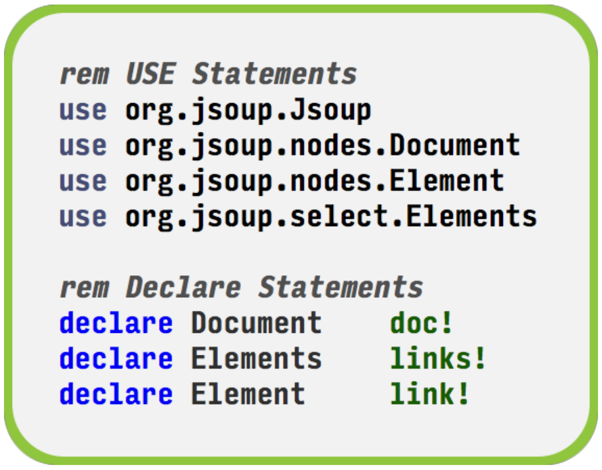
Using USE and DECLARE Statements

The Java example starts with `import` statements, shown in **Figure 8**, which are similar to BBj's USE Verb.

```
import org.jsoup.nodes.Document;
import org.jsoup.nodes.Element;
import org.jsoup.select.Elements;
```

Figure 8. An excerpt from the example jsoup Java program

By including a similar `use` statement for each `import` statement, we are able to reference the jsoup Java classes without having to specify the full package name of the class. That means that we can DECLARE our variables as a `Document`, for example, instead of declaring them as an `org.jsoup.nodes.Document.`
Figure 9 shows the first portion of our BBj version of the links example.



```
rem USE Statements
use org.jsoup.Jsoup
use org.jsoup.nodes.Document
use org.jsoup.nodes.Element
use org.jsoup.select.Elements

rem Declare Statements
declare Document    doc!
declare Elements    links!
declare Element     link!
```

Figure 9. An excerpt of our BBj links program - `jsoup.bbj`

Looking at the code, we start with a section of `use` statements that mirror the Java program's **`import`** statements. When we first confirmed that code completion was working in **Figure 7**, BDT showed matching completions based on what we had typed. This also happened when we typed in the `use` statements in **Figure 9**, and sometimes it is easier to select an item from the list rather than continue typing the rest of the line. An animation of BDT's code completion while typing a `use` statement is shown in **Figure 10**.

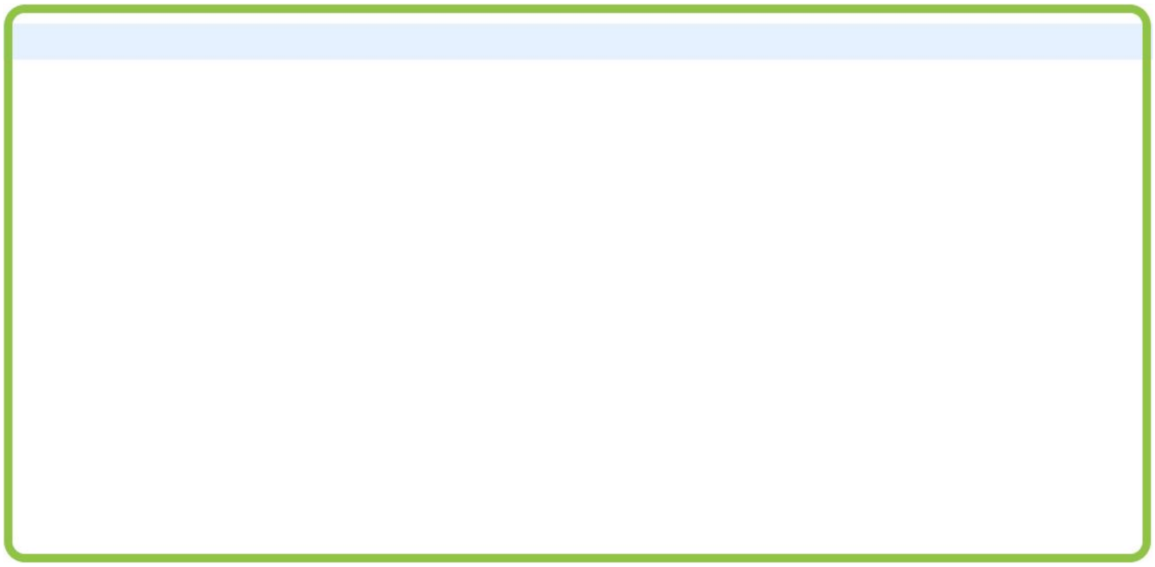
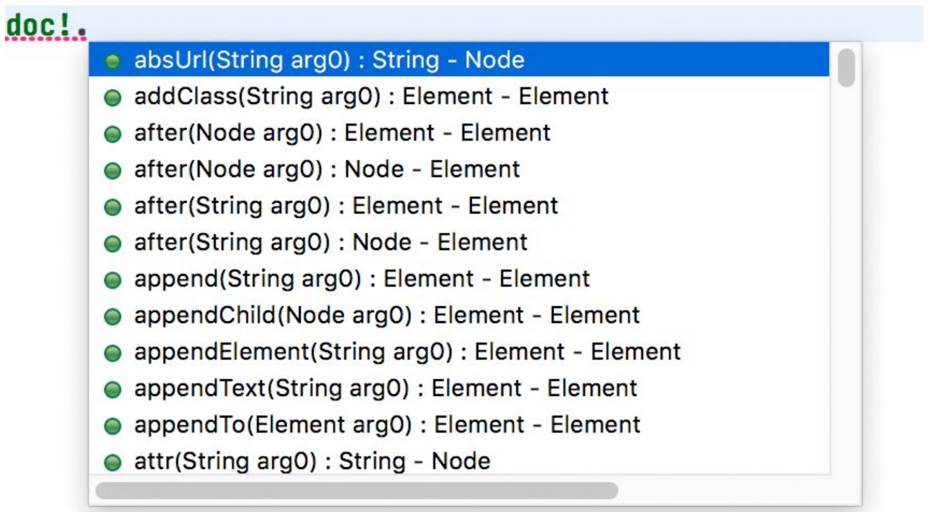


Figure 10. BDT's code completion for the program's `use` statement

The next block of code previously shown in **Figure 9** declares three object variables. Taking the first line as an example, we are telling BBj that the `doc!` variable's type will be an instance of the `org.jsoup.nodes.Document` class. Thanks to the previous `use` statement, we can shorten that to **`Document`** and BBj and the BDT will figure it out. Because everyone now knows that the `doc!` object is a **`Document`** instance, the BDT can give us code completion for that object's available methods. As a test, we can trigger code completion for that variable, as shown in **Figure 11**.



```
doc!.
● absUrl(String arg0) : String - Node
● addClass(String arg0) : Element - Element
● after(Node arg0) : Element - Element
● after(Node arg0) : Node - Element
● after(String arg0) : Element - Element
● after(String arg0) : Node - Element
● append(String arg0) : Element - Element
● appendChild(Node arg0) : Element - Element
● appendElement(String arg0) : Element - Element
● appendText(String arg0) : Element - Element
● appendTo(Element arg0) : Element - Element
● attr(String arg0) : String - Node
```

Figure 11. Triggering the code completion for the `doc!` variable

BDT shows us all the methods that apply to any **`Document`** instance in a searchable list. Every method shown is detailed in the **`Document`**'s Javadoc page, regardless of which class the methods come from. Note that the code completion popup also includes methods inherited from parent classes such as **`Node`**—basically any possible method that is valid for that type of object.

Porting the Rest of the jsoup Program

The next part of the example Java program builds a **`Document`** instance by executing the static **`connect()`** method on the **`Jsoup`** class, which is the core public access point to the jsoup functionality. **Figure 12** shows how it uses method chaining to take the result of the **`connect()`** method (a **`Connection`** object) and execute the **`get()`** method against it. We could have completed the operation in two separate steps, but sometimes it is more succinct to chain methods when you are programming in an object-oriented programming language like Java or BBj.

```
Document doc = Jsoup.connect(url).get();
Elements links = doc.select("a[href]");
```

Figure 12. The Java code to build a jsoup document and find all hyperlinks

The second line of the Java program finds every hyperlink in the document and loads them into an **`Elements`** class that extends a Java **`ArrayList`**. While not exactly the same, it is similar to a BBjVector which implements a Java List, providing a collection of webpage hyperlinks.

When we convert that Java code to BBj, most of the code will look about the same with a few exceptions:

1. Variable declarations—we have already declared our variables earlier in the program so we don't need to preface the **`doc`** and **`links`** variables with their type like the Java program does.
2. Semicolons—statements in Java use a semicolon to denote the end of the line. We don't need to do that in BBj, so one of the steps in porting the program is to remove the semicolons at the end of lines.
3. Object variable suffix—BBj object variables require a **!** suffix, so we have to change the **`doc`** and **`links`** Java variables to **`doc!`** and **`links!`** for the BBj version.

After making those changes, along with defining our URL in a separate line as `url$`, our BBj code is shown in **Figure 13**.

```
url$ = "https://www.basis.com"
doc! = Jsoup.connect(url$).get()
links! = doc!.select("a[href]")
```

Figure 13. The BBj code to build a jsoup document and find all hyperlinks

We have ‘written’ surprisingly little code so far, partly because we are converting an existing Java program, but also because the BDT’s code completion types in a percentage of the program for us. Regardless, running those few lines of code causes BBj to make a connection to the basis.com website, load the underlying HTML document, then scan it for hyperlinks that it stores in a collection variable. That is pretty significant for just three lines of code!

We can execute another line of code, as shown below, to print out the total number of hyperlinks on the basis webpage:

```
? links!.size()
```

At the time of this publication, there are 104 hyperlinks on that main page. If we wanted to access the individual links programmatically and print them to the BBj SysConsole, we could add a few more lines of code as shown in **Figure 14**.

```
for i = 0 to links!.size()-1
    link! = cast(Element, links!.get(i))
    print "link", i , ": """", link!.text(), """" = '", link!.attr("href"), "'"
next
```

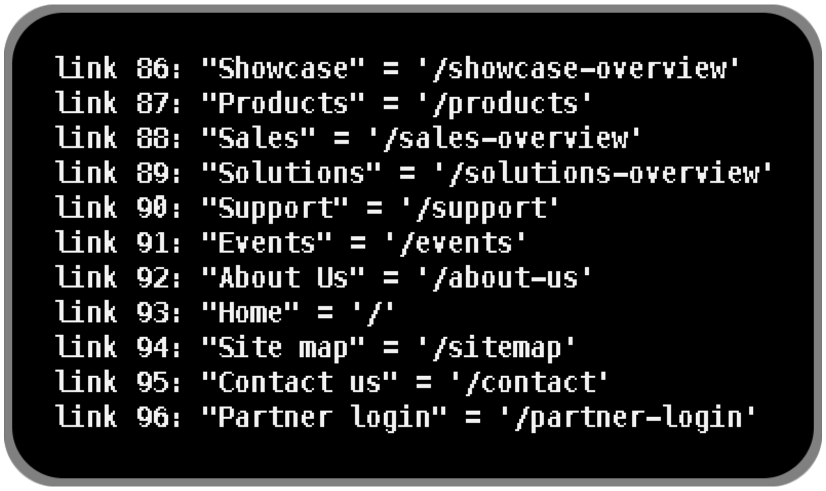
Figure 14. Printing out the list of hyperlinks to the BBj SysConsole

The code uses a BBj FOR/NEXT loop to iterate over all of the links stored in the `links!` collection. While we are typing the first line of code, BDT pops up the list of matching methods just after we finish typing in the `links!` variable followed by the period. As we start to type the `size` method name, the list of matching methods shrinks based on the letters following the period.

The second line in particular is interesting, as it uses the `CAST()` Function. Our code previously declared the `link!` variable to be a jsoup `Element`. Given that, why are we forcing it to be an `Element` with the `cast()`? We do that because we have to ensure that whenever we assign a value to `link!` we are doing so with an `Element` object. Assigning any other type, or an unknown type, to `link!` would result in an error. BBj does not know what type of object exists in the collection and it sees those items as generic `java.lang.Object`s. So we use the `cast()` function to convert the `Object` in the collection to an `Element` object. Alternatively, we could have included the `auto` parameter in our `declare` statement. This instructs BBj to automatically `cast` the variable if the value on the right hand side of the assignment is compatible with the declared variable type.

The `print` statement makes use of two more methods available to our `link!` variable. For this line, we opted not to convert the Java code line-by-line but instead chose methods displayed in the code completion popup based on their tell-tale names. As is often the case with object-oriented programming, well-named methods go a long way to documenting the class's abilities. By perusing the methods in the code completion popup, we were able to figure out that the **`text()`** method was going to be useful in our example. The **`attr()`** method is not quite as obvious, but the original Java example used that method in a similar manner so its usefulness was already apparent.

Executing that new code results in BBj printing over a hundred hyperlinks to our interpreter screen, with a portion of them shown in **Figure 15**.



```
link 86: "Showcase" = '/showcase-overview'
link 87: "Products" = '/products'
link 88: "Sales" = '/sales-overview'
link 89: "Solutions" = '/solutions-overview'
link 90: "Support" = '/support'
link 91: "Events" = '/events'
link 92: "About Us" = '/about-us'
link 93: "Home" = '/'
link 94: "Site map" = '/sitemap'
link 95: "Contact us" = '/contact'
link 96: "Partner login" = '/partner-login'
```

Figure 15. An excerpt of the hyperlinks printed to the BBj SysConsole

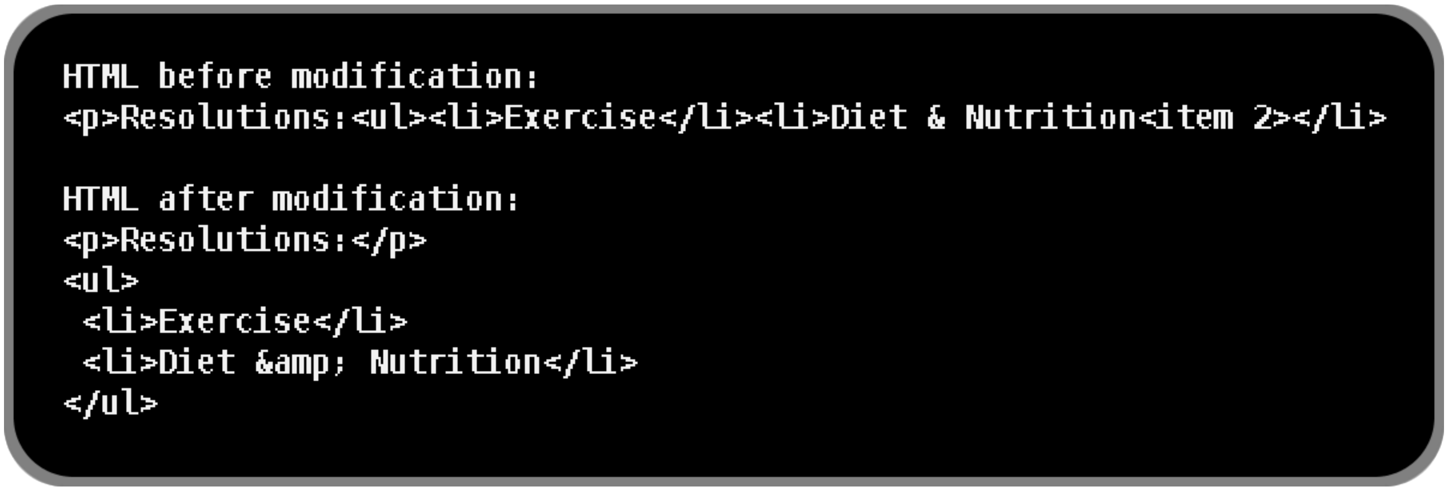
Writing Our Second jsoup Program - *tidying up HTML*

As mentioned earlier in the introduction, the jsoup library also has the ability to resolve missing closing tags, escape HTML entities, and output formatted HTML. This is easy to do if we use the jsoup **`clean()`** method. Besides formatting the document, it also strips out all extraneous elements that do not exist in a configurable Whitelist, sanitizing the HTML and making it resistant to cross-site scripting attacks. To keep our example simple, we will use one of jsoup's built-in whitelists. **Figure 16** shows the code in our new **`tidy.bbj`** program.

```
rem USE Statements
use org.jsoup.Jsoup
use org.jsoup.safety.Whitelist

rem Start with an HTML string that isn't formatted,
rem is missing a paragraph and an unordered list end tag,
rem and uses an un-escaped ampersand
messyHtml$ = "<p>Resolutions:<ul><li>Exercise</li><li>Diet & Nutrition<item 2></li>"

rem Clean and format the HTML with Jsoup
cleanHtml$ = Jsoup.clean(messyHtml$, Whitelist.basic())

rem Print out the before and after
? "HTML before modification: ", $0d0a$ + messyHtml$ + $0d0a$
? "HTML after modification:  ", $0d0a$ + cleanHtml$ + $0d0a$
```

Figure 16. A program to format HTML and resolve missing closing tags - **tidy.bbj**

This program is less complicated than our first sample. Because we employed jsoup's static `clean()` method, we did not need to instantiate any objects or use any BBj object variables. Instead, we can format our HTML text in a single line of code using traditional BBx string variables. The first string variable, `messyHtml$`, contains the source HTML text. In addition to all the tags being strung together on the same line, the HTML is missing closing tags for the initial paragraph (`</p>`) and the unordered list (`</ul>`) and uses an unescaped ampersand. The program cleans the contents of the `messyHtml$` variable, placing the results in the `cleanHtml$` variable. It then prints out the HTML string before and after modification for comparison, as shown in **Figure 17**.



```
HTML before modification:
<p>Resolutions:<ul><li>Exercise</li><li>Diet & Nutrition<item 2></li>

HTML after modification:
<p>Resolutions:</p>
<ul>
 <li>Exercise</li>
 <li>Diet &amp; Nutrition</li>
</ul>
```

Figure 17. Comparing the HTML source before and after formatting

The most obvious change is that our HTML source is now pretty-printed, making it easier to understand and maintain. Besides the formatting changes, notice that jsoup wrapped the paragraph element by adding the closing tag before the unordered list. It also closed out the list element by adding the missing closing tag and escaped the reserved ampersand character to avoid misinterpretation.

Conclusion

This article explored integrating jsoup, a free third-party Java library, into several facets of development including BBjServices' classpath, our BBj programs, and even our BDT development environment. While it takes a little effort to configure, as covered in the four steps in the *Reviewing Our Progress* section, we end up saving time and effort in the long run. By taking advantage of pre-tested, well-maintained existing code, we spend less time reinventing the wheel—especially when it comes to a common necessity like parsing text. The jsoup library not only simplifies our BBj programs by allowing us to leverage abstracted method calls, it also modularizes our program by keeping the library's code separated from our application. The two communicate via well-defined APIs, so if the need should ever arise we could decouple it from our application code without excessive effort. Best of all, we get all the benefits of a software library written by developers who are likely more experienced than ourselves in that particular area of code.

External libraries do have potential drawbacks, though, as they may have dependencies on several other libraries. All the libraries that you intend to use should preferably be well-supported, actively maintained, appropriately licensed, and updated on a regular schedule. Following common-sense guidelines can save you tons of development time, buy you subject matter expertise, and sometimes even surprise you with new features that magically appear after an update.

Recommended Reading

A variety of material is available online for developing BBj programs in the Eclipse IDE. Along with general documentation, BASIS offers additional Advantage articles, white papers, and other resources that specifically cover object-oriented programming and working with Java code and JARs. Lastly, the IDE User Group Wiki extends these offerings with more documentation, tutorials, and sample code. Many of these resources are linked below for easy access and further study.

- View or download the **`jsoup.bbj`** program (first program)
- View or download the **`tidy.bbj`** program (second program)
- A Primer for Using BBj Custom Objects - An overview of BBj custom classes
- BBj Custom Objects Tutorial - An in-depth whitepaper on BBj custom objects
- Session-specific Classpath Documentation - SSCP online documentation
- Eclipse BDT Overview Documentation - The root BDT online documentation
- Have it Your Way With New BDT Preferences and Properties - An overview of the BDT configuration, preferences, and settings
- BDT Tips for Less Pain and More Gain - A collection of time-saving tips and other productivity enhancements for those developing with Eclipse and the BDT plug-in

Eclipse IDE Section in the IDE User Group Wiki

The Wiki allows developers to share resources and access documentation, sample code, or other development artifacts provided by the community. It contains several sections devoted to the BASIS IDE, the BDT Plug-In, and other applications and utilities such as the Dashboard Utility and BBJasper. Contact BASIS Sales by emailing info@basis.com for an account and instructions on how to get started accessing tutorials and contributing your own code.

- IDE User Group Wiki