# Getting on the Right (Class)Path

*By Chris Hardekopf and Jerry Karasz*

## Classpath Overview

BBj's use of Java classpaths became more flexible and powerful than ever before with the release of BBj® 19.10. You now have the ability to control virtually every aspect of how your BBj programs find and use Java classes.

But before we start digging into how classpaths have changed, let's take a moment and talk about what a classpath is, why you use them, and why you care about having more control over them.

### What is a classpath?

A classpath is essentially an ordered collection of places for the Java interpreter to search when it needs to find the definition of a class. Usually, it's an ordered list of Java ARchive (JAR) files, each of which contains one or more Java class definition files.

Please note: Classpaths have nothing to do with BBj's PREFIX! Classpaths are instrumental for finding Java class definitions, while the PREFIX is for finding BBj programs and other resource files such as images and data.

### What is a classloader?

Once Java searches the appropriate classpath(s) and finds a class your code needs, the next step is to load that class information into the Java Interpreter (so that it can be executed). This process requires a Java classloader, which is basically an object that is responsible for loading Java class definitions so that your interpreter can run them.

## What does that mean to you?

Since BBj is written in Java, it has always had a “Default Java Classpath” that it used to specify the JAR files it needs to run. Like it or not, your BBj program had that same classpath, too.

What could you do? Well, you could append other classes and JAR files to that classpath by defining a Session-Specific Classpath (SSCP). But you could not override or preempt the version, or the order, of any of the classes that BBj needed to run. Additionally, you could not put any JAR files you wanted to use ahead of BBj’s JAR files.

For example, if BBj Version X included the version Tika_Y.Z.jar, you could not use an older OR a newer version of Tika in your programs. Your options were limited.

## Why do you care?

It’s possible you never noticed these JAR dependency limitations. But, even if you were happy using exactly the same JAR files that BBj used, you may have noticed a few problems:

- Whenever BBj was released with a newer version of a JAR file, your programs were required to run with that newer version, too.
- If BBj continued using an older version of a JAR file, you were unable to upgrade to a newer version.
- If you wished to build a modified version of an OpenSource JAR file and use it instead of BBj’s included version, you couldn’t.
- Any time BBj changed the Default Java Classpath, there was a possibility that your program might break if it had dependencies on the prior Default Java Classpath.

# The New Classpath Paradigm

BBj 19.10 introduced a new classpath paradigm, in which the classpath that BBjServices uses internally is completely independent of the classpaths that you can configure and use in your BBj applications.

The new functionality allows you to specify the exact JAR that you want your BBj program to use, regardless of the version BBj installs and uses internally. You no longer need to change to a different version just because BBj decided to use an older or newer version. You also do not need to be bothered when BBj adds a JAR file to its classpath that conflicts with one that you were already using.

Starting with version 19.10, BBj comes pre-configured with a number of classpaths. These classpaths help you manage any dependency you may have on BBj’s internal list. Let’s take a look at the two most important of these paths: **bbj_internal** and **bbj_default**.

## bbj_internal

The **bbj_internal** classpath contains all of the JAR files that BBj itself needs to run. You can use this classpath to continue accessing the list of JAR files that BBj requires, just as you have done in the past. **bbj_internal** is a read-only classpath. It cannot be edited manually, but it will be updated internally as necessary (as it has always been).

## bbj_default

The **bbj_default** classpath replaces the old “Default Java Classpath”. As its name indicates, it is the default classpath that BBj uses whenever you run without specifying your own SSCP.

**bbj_default** is pre-configured with exactly one entry: the **bbj_internal** classpath. With this configuration, you can continue to run exactly as you always have: BBj will maintain that list of JAR files for you, and each time you run BBj without specifying your own SSCP, BBj will use the **bbj_default** classpath.

But, unlike the old “Default Java Classpath”, you can edit **bbj_default** as you wish. You can add, reorder, or remove any and all entries. You can even remove the **bbj_internal** entry and be completely free from BBj’s JAR file list.

## The Good News:

You are in control of your Java classpath!

You can choose what will be in your classpath. You can keep your classpath as you have defined it, without anyone else changing it. You can even reference the new **bbj_internal** classpath in your own classpath if you choose (read on for details).

## The Bad News:

You are in control of your Java classpath!

Once you define an SSCP, you must maintain that SSCP for your programs to use. You will have to decide when you are ready to update any of the JAR files you use. You will also have to decide when to add or remove JAR files from it.

## More Good News:

Your classpaths can now inherit from other classpaths!

Similar to class inheritance, your classpath can now “inherit” from a parent classpath. Remember how we said you can still use BBj’s **bbj_internal** classpath if you like? Well, your classpaths can ***inherit*** from **bbj_internal** – if you wish.

Here are a few things to know about inheriting classpaths:

- The parent classpath always comes “first” in your classpath.
- A parent classpath takes priority in loading classes (if there is a duplicate entry in the parent classpath and in the child classpath, the parent’s entry will win).
- If the parent classpath can find and load a class, then it will do so before it makes any attempt to even look at the child classpath.

## Even More Good News:

Your classpaths can include other classpaths directly!

Instead of having your classpath inherit from another classpath, you might prefer to simply “include” another classpath in yours. Remember how we said you can still use BBj’s **bbj_internal** classpath if you like? Well, your classpaths can ***include*** **bbj_internal** – if you wish.

With this approach, you have more control over the order of the included JAR files (by including the other classpath at any point you choose in your classpath’s list of JAR files). Here are some of the advantages and differences:

- You can include another classpath at any point in your classpath; this means that you can specify JAR files or other classpath entries to search before or after the included classpath, as you like.
- At runtime, BBj expands the included classpath to a list of JAR files, and includes those JAR files directly in your classpath.
- Including classpaths in this way lets you share a list of JAR files without duplicating the entire list; updating that classpath will update any and all classpaths that include it.

## But just in case...

If you do not wish to manage your own SSCPs, you can simply run without specifying your own SSCP. BBj will then use the **bbj_default** classpath, and will run as it always has – using the list of JAR files that BBj maintains for itself (and for you).

Remember, unlike the old “Default Java Classpath”, you can edit **bbj_default**. You can add, reorder, or remove any and all entries. You can even remove the **bbj_internal** entry, and be completely free from BBj’s JAR file list.

# Getting Started

By now, you may be asking yourself, “How do I get started?” All you have to do is create your own Session Specific Classpath (SSCP) and tell BBj to use it when it runs.

## How do you create your own SSCP?

You can create and manage any number of SSCPs in the Enterprise Manager’s (EM’s) Classpath tab, as shown in **Figure 1**.

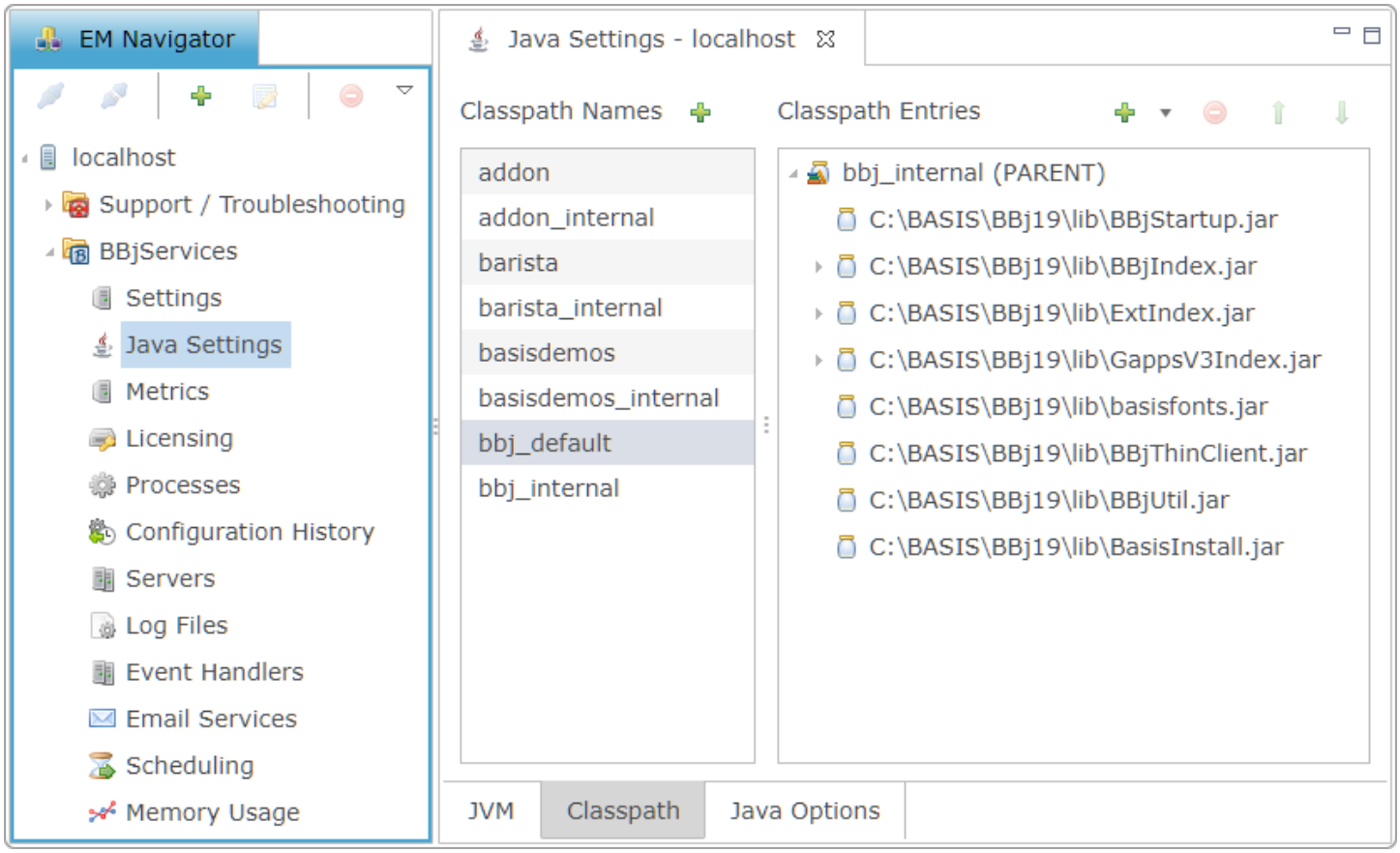


**Figure 1.** Java Classpaths in the Enterprise Manager

Notice that several classpaths are automatically created for you. These are visible in **Figure 1** above. The list of pre-configured classpaths may change over time, but the two we discussed earlier are **bbj_default** and **bbj_internal**. Notice that you can expand any classpath in the EM to see exactly what JAR files it contains. Note that the pre-configured classpaths ending in “_internal” are not editable.

 For a full explanation of how to manage your SSCPs in the EM, see the EM help page.

## How do you use your SSCP?

Once you have created at least one SSCP, you can tell the BBj interpreter to use it as your session's classpath by using the "-CP" command line option. For example, if "MyClasspath" is the name of your classpath, use the following command line:

```
bbj.exe -CPMyClasspath <program>
```

Alternatively, if you do not use a command line option, BBj will use the **bbj_default** classpath, which is fully under your control and is pre-configured to offer you backwards compatibility.

# Summary

With BBj's new classpath paradigm, you are in charge. You can:

- dictate that your application uses a newer, or an older, version of a Java class from the one that BBj relies on
- control virtually every aspect of how your BBj programs will interact with Java
- choose to leave that control in BBj's hands, as most users have in the past
- choose to completely redefine how Java classes are found and loaded – including removing all references to BBj's internal Java class list if you wish

BASIS continues to strive toward delivering an ever expanding rich set of development tools for creating new business applications or enhancing existing applications. With BASIS' new Java class selection capability, your choices of Java libraries have expanded dramatically and will allow you more freedom to incorporate Java into your solution as and when needed.