WebSphere Application Server for z/OS

Writing Plug-ins for the WebSphere Application Server SMF Record Browser

This document can be found on the web at:
www.ibm.com/support/techdocs
Search for document number **WP101726** under the category of "White Papers"

Version Date: January 12, 2022
See "Document change history" on page 17 for a description of the changes in this version of the document

IBM Software Group
Application and Integration Middleware Software

Written by David Follis
IBM Poughkeepsie
845-435-5462
follis@us.ibm.com

Many thanks go to ...Don Bagwell, John Hutchinson, and Mike Loos.

Introduction

The Whitepaper WP101342, “Understanding SMF Record Type 120, Subtype 9” discusses the new WebSphere SMF record in detail. That paper also briefly discusses updates to the SMF Browser provided by WebSphere development. One of these updates provides the ability for you to write plug-ins to perform your own analysis and reporting. While many customers have their own tools to process and analyze SMF records, some customers may find it convenient to be able to write quick little Java programs to do some basic things. This paper goes into more detail about the mechanics of writing plug-ins for the browser and provides the source for a simple example you can use to get started.

You might also look at WP102311 and WP102312 for some plugins I've written and a look at what you can learn using them.

Getting and Using the Browser

Update – January 2022

In mid-2020 the SMF browser and the plugins described in WP102312 were moved to open source in github (https://github.com/IBM/IBM-Z-zOS/tree/main/SMF-Tools). The browser itself was broken into two .jar files (SMF_CORE and SMF_WAS). Usage documentation can be found in the ReadMe. Note that package names changed, so the command examples below won't work exactly as-shown. Going forward only the github browser will be updated. The 'old' one will stay where it is, for now anyway.

The browser can be downloaded from this URL: https://www.ibm.com/services/forms/preLogin.do?source=zosos390&S_PKG=smf5

If you do not already have an id, you can easily register and create one. When you get to the download page, you should be able to select “SMF Browser for WebSphere Application Server for z/OS V5, V6, and V7”. Note that it also works for V8 and above. It turns out to be very easy to replace the .jar file with an updated one, but very complicated to get the actual text of the web page changed.

The result of the download will be the file bbomsmfv.jar. You will want to copy bbomsmfv.jar to a directory in your USS file system on z/OS. The .jar file includes the executable, the source code for the browser, some documentation, license material, and a change history file.

After you have the bbomsmfv.jar file in your USS file system, you can use the .jar file to view the SMF 120 records in an SMF dump dataset. For the rest of this paper we will assume you have run some WebSphere requests through a Version 7 server with SMF type 120, subtype 9, recording enabled and dumped the SMF data into an MVS dataset called ‘WAS.SMF.DATA’.

To run the browser you need to first set the classpath. The browser is dependent on another .jar file to read the MVS dataset where the SMF data exists. That .jar file is called ibmjzos.jar. The .jar file should be part of any z/OS JVM you have installed (there is a copy in the Java lib/ext directory where WebSphere is installed).

If both .jar files are in your current directory, this command will run the browser with the default plug-in (on one line):

```
java -cp bbomsmfv.jar:ibmjzos.jar com.ibm.ws390.sm.smfview.SMF
"INFILE(WAS.SMF.DATA)"
```

You might consider putting that into a file, we will call the file smf.sh, and use chmod to make the file executable with the command:

```
chmod 700 smf.sh
```

Now you can just type smf.sh to run the browser.

Writing Your Own Plug-ins

There are some different approaches to creating your own plug-ins.

Adding to the browser jar file

You can import the bbomsmfv.jar file into your IDE (such as Eclipse) and just add things to the file. You can add to the com.ibm.ws390.sm.smfview package or create your own. This approach is certainly the easiest. On the other hand, incorporating your changes into bbomsmfv.jar will make the process of incorporating future updates from IBM harder.

Using your own jar file

Another alternative is to make your own .jar file. You can just include the bbomsmfv.jar file as an external jar to be used to resolve references by the java compiler. This approach keeps your code and the IBM code separate and is generally the preferred approach.

However, unless you have a recent build of the browser (look for dates in the change history near the publication date of this paper), you might find using this approach to be difficult to do. As part of writing this paper, we made some changes to the browser to make writing plug-ins in separate .jar files a little easier.

Our Example

The purpose of this paper is to show you how to write your own plug-ins. However, the example provided should also serve some useful purpose. Hopefully we will meet both goals.

Many customers run WebSphere servers with more than one servant region. Those customers often wonder how work is being distributed across those servant regions. There are several ways to tell and the SMF 120-9 record content is one of them. All we need to do is examine each record to determine which servant dispatched the request and summarize our results.

Therefore, the goal of this example will be to produce a report listing the servers and servant regions for which we found records along with a count of the number of requests dispatched in each servant. As a bonus we will try to break down the counts by type (e.g. http, IIOP, etc.). We will be creating output that looks like this:

```
Server   JobID    ASID  Unknown IIOP    HTTP    HTTPS   MBean   OTS     Other
Total
BBOS001  STC00044 003b  0       0       138     71      1       0       47     257
-------- -------- ----  ------- ----    ----    -----   -----   ---     -----
-----
Totals                  0       0       138     71      1       0       47     257
```

Getting Started

You can create and build Java programs without a development environment, but development is a lot easier with one. For our examples we will assume you are using the base Eclipse IDE. If you

do not have some other IDE installed and have never used Eclipse, we will try to get you started with a few quick directions here.

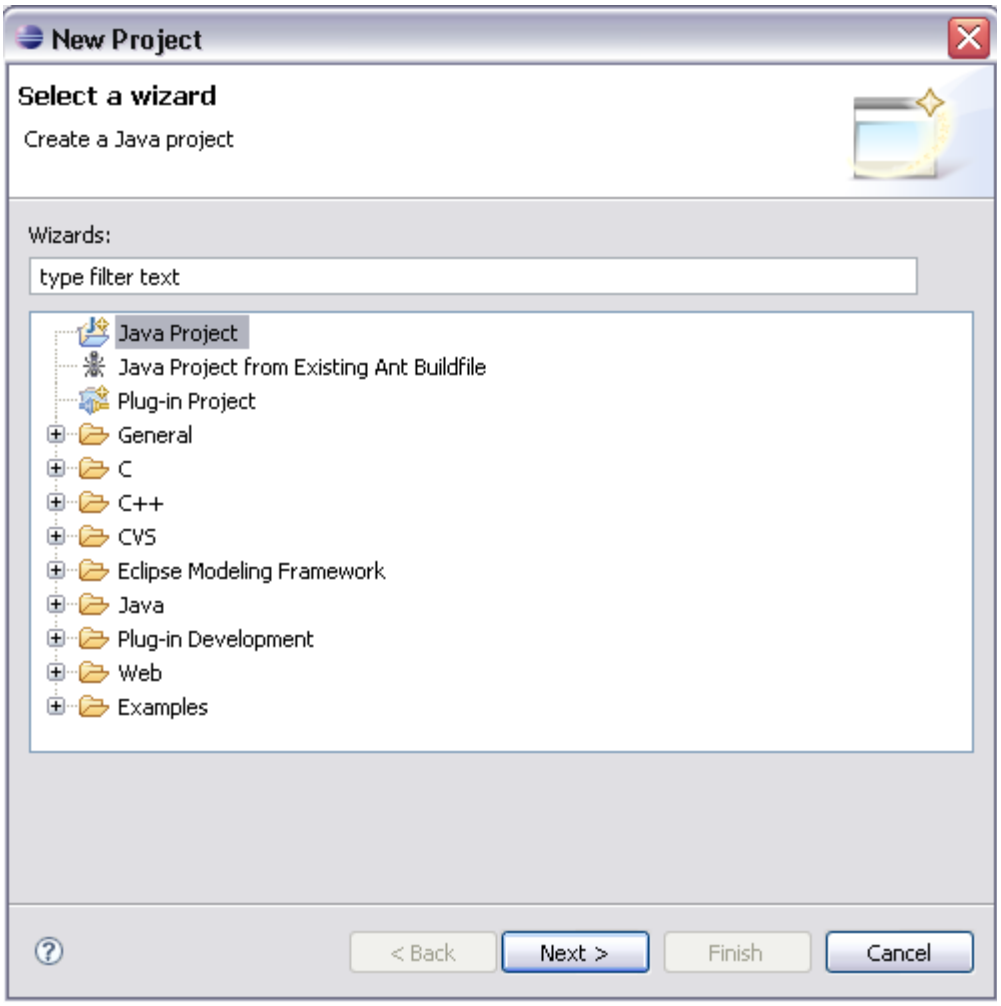
Getting Eclipse

Eclipse can be obtained from http://www.eclipse.org/downloads.

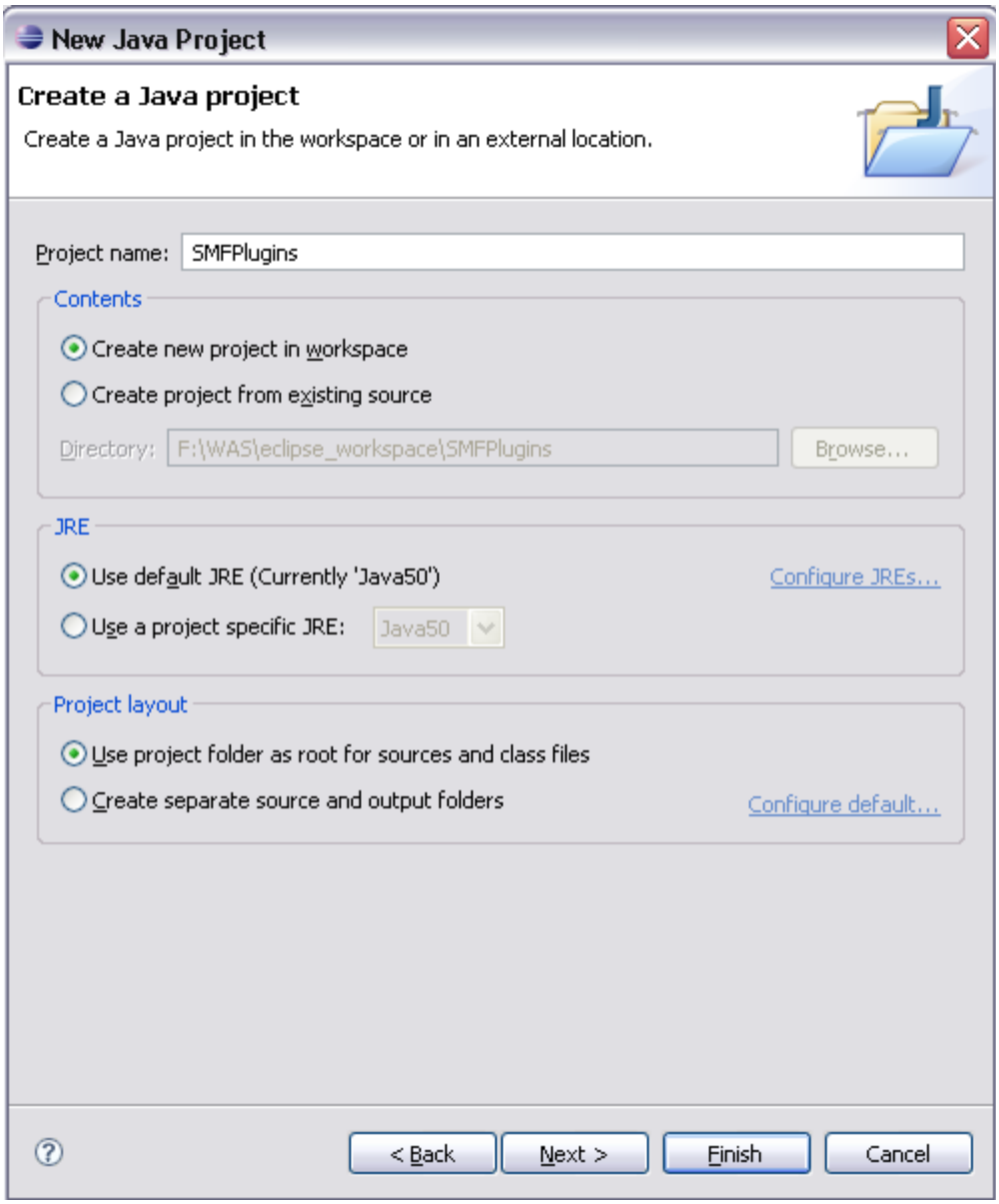
Creating a Project

After you have Eclipse installed you need to create a project for your plug-ins. Here are the steps:

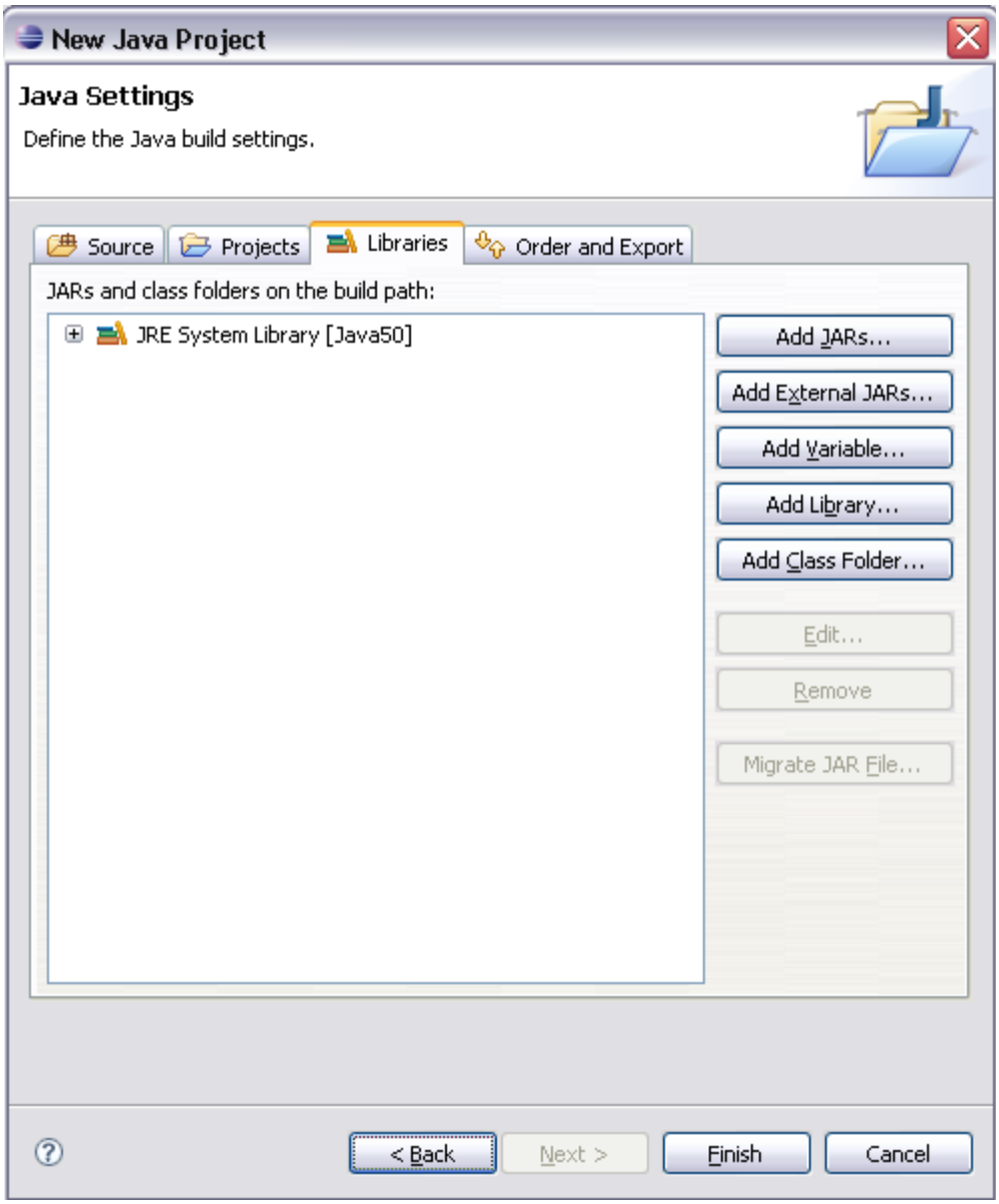
1. Select 'File' -> 'New' -> 'Project' to start a wizard to help you with the project creation.
2. Because you want to create a Java project, make sure that is selected and click on 'Next'.



3. Enter a project name in the box, for example ‘SMFPlugins’. Click ‘Next’.



4. Now we need to get the supporting .jar files. Select the 'Libraries' tab.



5. Click on 'Add External Jars' to bring up a file browser.
6. Find and select the bbomsmfv.jar and ibmjzos.jar files discussed earlier.
7. Click on 'Finish'

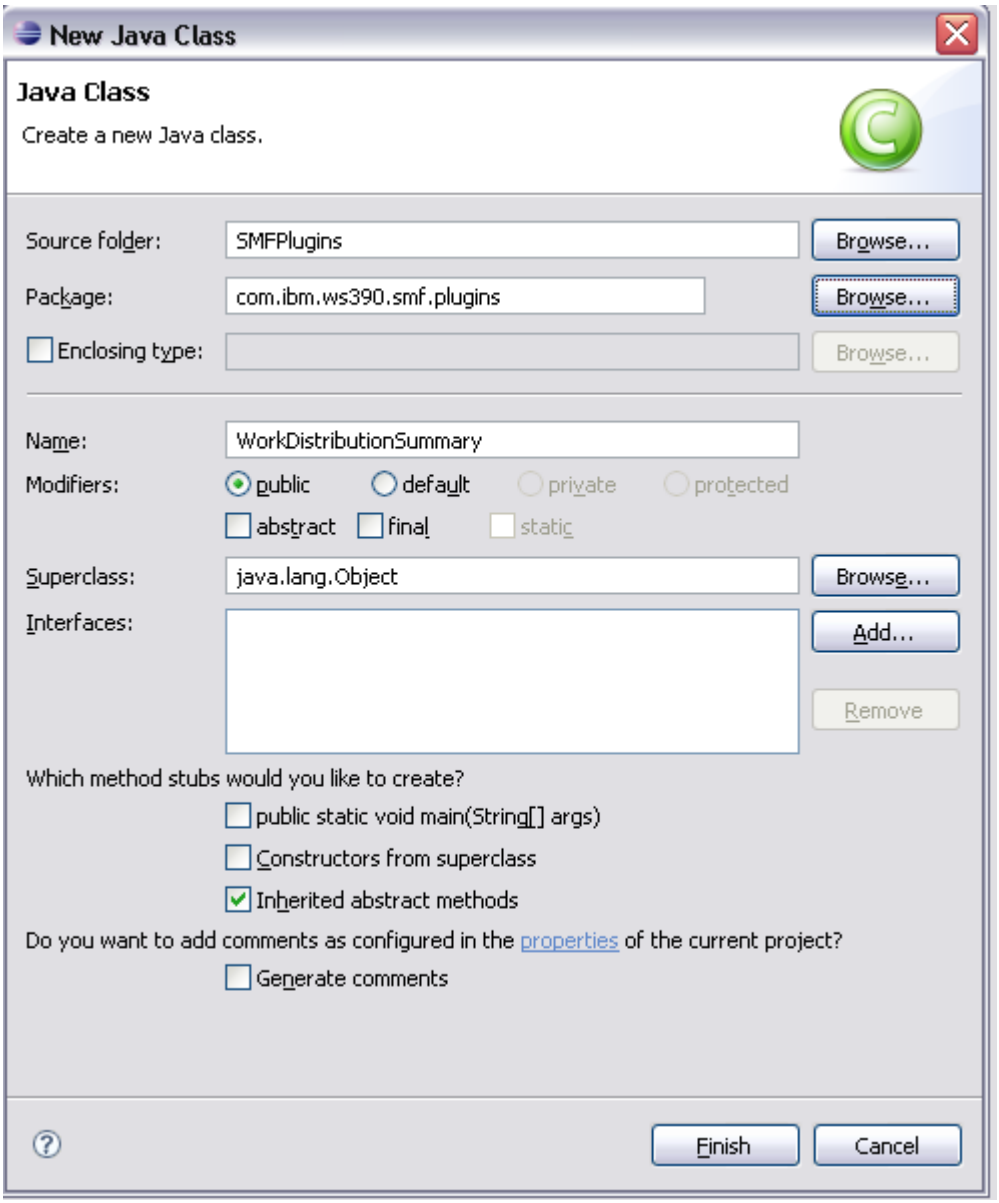
Now you should have the SMFPlugins project listed on the left side of the screen. If you expand the section you should see the two .jar files we added as build pre-requisites.

Defining the Plug-in class

The next step is to define a Java class to contain your plug-in. With Eclipse, these are the steps:

1. Right click on the project you defined to bring up a menu
2. Select 'New' -> 'Class'
3. In the 'Name' field enter the name of your plug-in. For our example, we will use WorkDistributionSummary as the name.

4. Enter a package name (if you want one) in the field near the top. I will use com.ibm.ws390.smf.plugins. Use whatever you like.



5. Click on 'Finish'

Now you should see a section under the SMF Plugins project with your package name. Expanding the section should show you the class you just defined. Eclipse will open the new class file in the editor section. We are going to need to import some classes from the bbomsmfv.jar file into our new class. We could have added those import statements as part of defining the class with the wizard, but I will leave adding imports for later as we go through each section of the code.

Writing the Required Methods

This section will go through the code in the plug-in a little at a time.

Before we start writing the code we need to import some classes we will need and set the inheritance of our class. As mentioned above, we could have imported classes via the wizard when we defined the class, but for this example we will just cut and paste the code in.

First of all, we need to import things. You should add this list to your Java class after the package statement but before the declaration of the class:

```
import java.util.HashMap;
import java.util.Iterator;
import com.ibm.ws390.sm.smfview.DefaultFilter;
import com.ibm.ws390.sm.smfview.PlatformNeutralRequestInfoSection;
import com.ibm.ws390.sm.smfview.PlatformNeutralSection;
import com.ibm.ws390.sm.smfview.RequestActivitySmfRecord;
import com.ibm.ws390.sm.smfview.SMFFilter;
import com.ibm.ws390.sm.smfview.SmfPrintStream;
import com.ibm.ws390.sm.smfview.SmfRecord;
import com.ibm.ws390.sm.smfview.SmfUtil;
import com.ibm.ws390.sm.smfview.Triplet;
import com.ibm.ws390.sm.smfview.ZosRequestInfoSection;
```

We also need to change the definition of the class to implement the SMFFilter interface. This change will allow the class to be driven by the browser when the user asks for this class to be used as a plug-in. Update the class definition to look like this:

```
public class WorkDistributionSummary implements SMFFilter
```

Before we start coding, we need to declare some class attributes. We are going to need a place to keep the results as we go through the records. We define a HashMap to contain them. Because we need to follow the rules of the browser for printing output, instead of just using System.out.println, we will need to create and use an SmfPrintStream. We will declare a variable to hold a reference to the stream.

```
private SmfPrintStream smf_printstream = null;
private HashMap Servants = new HashMap();
```

Put these inside the class definition.

Now we are ready to start implementing the methods required by the SMFFilter interface. The following sections will take you through those methods one by one.

The initialize Method

The initialize method gets control as the browser is starting. This method is where you can parse any parameters that get passed in after the comma in the PLUGIN keyword. Usually this method is used to set up the destination for output and initialize variables. The default filter already does the 'standard' thing to set up an output destination. It is easiest just to let the default filter handle this processing. Here is the code:

```
public boolean initialize(String parms)
{
 boolean return_value = true;
 smf_printstream = DefaultFilter.commonInitialize(parms);
 if (smf_printstream==null)
      return_value = false;
 return return_value;
}
```

If the default filter has problems setting up the printstream a null will be returned. The initialize method has to return a Boolean indicating if initialization completed successfully or not. If a printstream was set up, our filter is all set.

The preParse Method

The preParse method gives you a peek at the SMF record before any attempt is made to parse the full record. Mostly this method is used to tell the browser to skip this particular record because the record is not interesting.

The SMF record header has been parsed and can be used to determine the record type and subtype. These values are important in determining if you care about this record. In our case we just want to look at type 120 subtype 9 records, therefore we will examine the type and subtype. Here is the code:

```
public boolean preParse(SmfRecord record)
{
 boolean ok_to_process = false;
 if (record.type()== SmfRecord.SmfRecordType)
   if (record.subtype()==SmfRecord.RequestActivitySmfRecordSubtype)
         ok_to_process = true;
   return ok_to_process;
}
```

The preParse method returns a Boolean indicator to tell the browser whether to continue with this record or not. We return true when the record is a type 120, subtype 9 record. Note that the SmfRecord class does not contain constants for all types and subtypes. You may have to make your own if you need to look at types other than the type 120 record.

The parse Method

The parse method can be used to parse apart the record into some structure you can use. The browser was built to parse SMF type 120 records and therefore already knows how to handle the only record type we are interested in. We will just let the default filter handle the parsing. Here is the code:

```
public SmfRecord parse(SmfRecord record)
{
 return DefaultFilter.commonParse(record);
}
```

The processRecord Method

This is where the action is. The processRecord method gets control after each record that has survived preParse has been parsed. At this point we know we have a type 120 subtype 9 record and we are free to roam around picking up whatever values we need. For our example we are going to collect four pieces of information: server name, servant ASID, servant jobname (e.g. STC12345), and the type of the request that was executed.

We will start by initializing some local variables to hold the information we plan to get from the record as well as a reference to a ServantData object in which we will be collecting our results for each servant. We will also cast the generic input parameter SmfRecord to the type 120 subtype 9 specific class, RequestActivitySmfRecord. Here is the code:

```
public void processRecord(SmfRecord record)
{
 int request_type= 0;  // default to unknown
 String servername = null;
 String jobID = null;
 String asid = null;
 ServantData sd = null;

 // cast to a subtype 9
 RequestActivitySmfRecord rec = (RequestActivitySmfRecord)record;
```

Next we need to see if this record includes an async data section. If it does, then this record is for async work run via a WorkManager. The section that includes the request type isn't in these records. We could also count async work in addition to the other types. For simplicities sake, we'll just ignore them. Here's the code:

```
      // Ignore Async work records...
      Triplet asyncWorkTriplet = rec.m_asyncWorkDataTriplet;
      int sectionCount = asyncWorkTriplet.count();
      if (sectionCount >0) {
            return;
      }
```

The first piece of information we will get is the server name. The server name is located in the platform neutral server information section. To find that section, first we need to get the triplet that locates the section. The RequestActivitySmfRecord has public attributes for each triplet. We can simply get the triplet.

Each Triplet contains a count. We examine the count to determine if the section is present or not. The platform neutral server information section should always be present, but a good practice is to check just in case.

If the section is present we can get a reference to the section itself from the record through another attribute called m_platformNeutralSection.

The server name is a public attribute of the PlatformNeutralSection, therefore we can get a reference to the name in our local variable.

```
 // get the Platform Neutral section
 Triplet platformNeutralTriplet =
rec.m_platformNeutralSectionTriplet;

 int platformNeutralSectionCount = platformNeutralTriplet.count();

 // if we have one, get the server name
 if (platformNeutralSectionCount>0)
 {
  PlatformNeutralSection PN = rec.m_platformNeutralSection;
  servername = PN.m_serverShortName;
 }
```

Next we need the request type. That value is kept in the platform neutral request information section. We use code very similar to what we used to get the server name, however this time we are after the PlatformNeutralRequestInfoSection. The variable names are a little terse in this case to prevent excessive line wrapping when pasted into this document.

```
// get the Platform Neutral Request section
Triplet pNRT = rec.m_platformNeutralSectionTriplet;

int pNRTCount = pNRT.count();

// If we have one...get the request type
if (pNRTCount>0)
{
 // get the request type
 PlatformNeutralRequestInfoSection PNRI;
 PNRI = rec.m_platformNeutralRequestInfoSection;
 request_type = PNRI.m_requestType;
}
```

Next we need the jobname and the ASID. These are in the z/OS Request section. As before, we find the triplet, make sure the count is not zero and fetch the data we want from the section. Because the ASID is stored as a byte array we use a utility function from the browser to convert the value to a string. Here is the code:

```
// get the zOS request section
Triplet zOSRequestTriplet = rec.m_zosRequestInfoTriplet;

int zOSRequestSectionCount = zOSRequestTriplet.count();

// If we have one, get the servant asid and jobname
if (zOSRequestSectionCount>0)
{
 ZosRequestInfoSection ZRI = rec.m_zosRequestInfoSection;
 jobID = ZRI.m_dispatchServantJobId;
 asid = SmfUtil.hexString(ZRI.m_dispatchServantAsid,0,4);
}
```

Now that we have all of the information we need from the record, we need to update our running totals. For each servant region for which we have data we will create a ServantData object to contain information about the servant. Therefore, we next need to determine if we have already created a ServantData object for this servant or if we need to make a new one. We will keep all of the ServantData objects in the HashMap called Servants.

First we look in the Servants HashMap for a ServantData object that has the same asid as the servant used for the record we are processing. If we find one, we are all set. If we do not find a ServantData object we create one, passing the server name, job name, and asid to the constructor. Finally we add the new ServantData object to the Servants HashMap. Here is the code:

```
// Seen this servant already?  If not make a ServantData object
if (asid!=null)
{
 sd = (ServantData)Servants.get(asid);
 if (sd==null)
 {
  sd = new ServantData(servername,jobID,asid);
  Servants.put(asid,sd);
 }
```

All that is left to do is update the ServantData object with the information from this request. In this case that is just the type of the request. The ServantData object will update a counter for this request type.

We are all done with the processRecord method!

```
 // Update ServantData with this request
 sd.countWork(request_type);
 }
}
```

The processingComplete Method

The processingComplete method is called when the browser has finished processing all of the records in the SMF dump dataset provided on the INFILE keyword. This method is your filter's opportunity to print out any conclusion or summary information.

In our case the information we want to report is all kept in the ServantData objects contained in the Servants HashMap. All we need to do is iterate over the contents of Servants and call the toString() method on each ServantData object to print the results. We will call static createHeader and createFooter methods to print a table header and footer. These methods are in ServantData instead of here because it is easier to keep the column alignment straight with all of the text creation in one place.

Taking this step by step, first we need to get an iterator from our HashMap and determine if there are any ServantData objects in the map. If we have data, print the header:

```
public void processingComplete()
{
 Iterator it = (Servants.keySet()).iterator();
 // Got anything?
 if (it.hasNext())
 {
  smf_printstream.println(ServantData.createHeader());
```

Now we iterate over the set of ServantData objects we got from the HashMap. For each ServantData object we call the toString method to get a string to print.

```
  while (it.hasNext())
  {
   // Get the asid from the iterator
   String asid = (String)it.next();
   //Get the ServantData object for this asid
   ServantData sd = (ServantData)Servants.get(asid);
   // Print the data from the Servant
   smf_printstream.println(sd.toString());
  }
```

And that is about it. All that remains is to print the footer string and end the method:

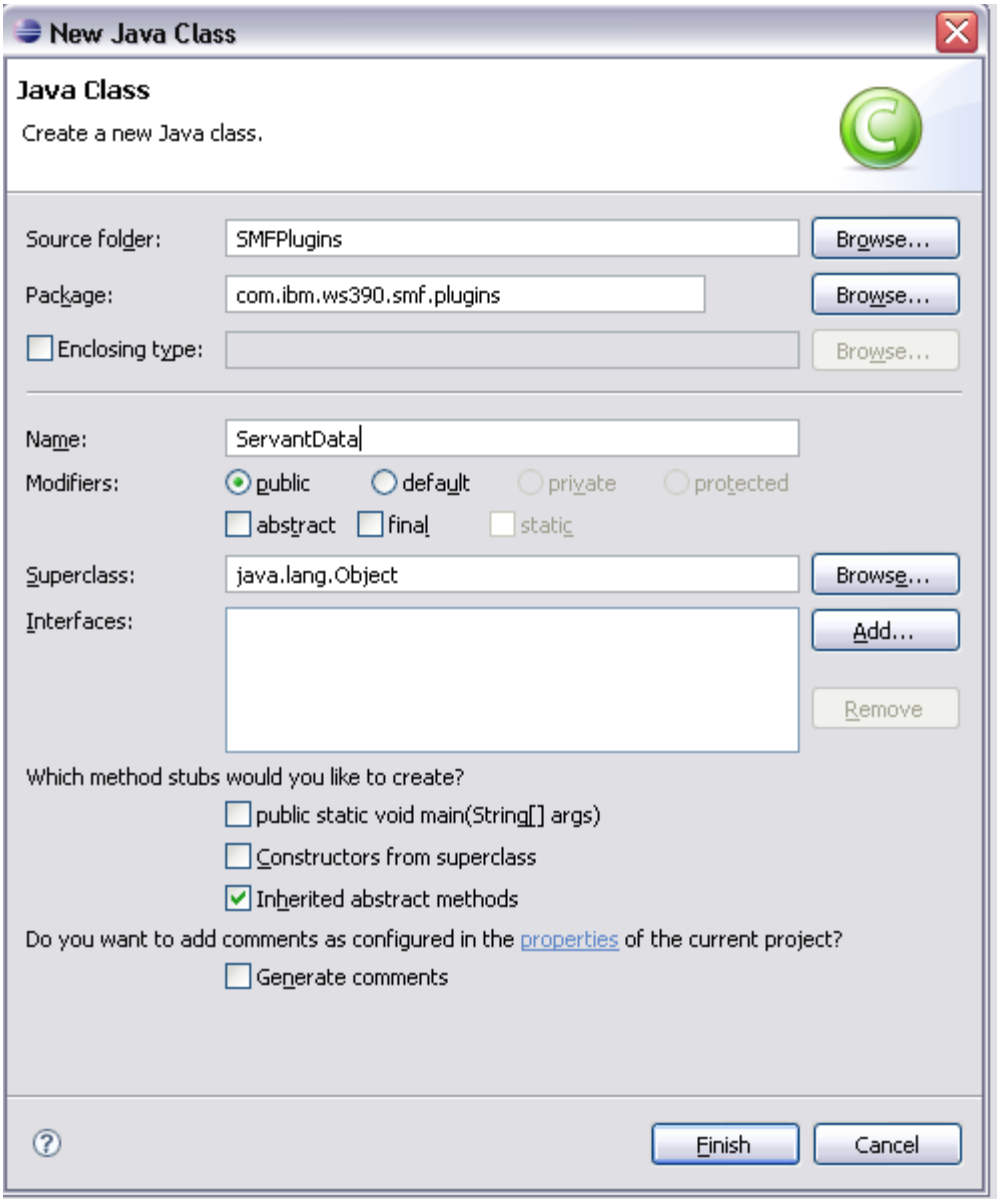
```
    smf_printstream.println(ServantData.createFooter());
  }
}
```

This concludes the description of the filter itself. From these examples you should be able to see how to fetch fields from the different sections of the SMF 120 records. To make that code easier to follow, we pushed a lot of the results management into the ServantData class. Next we will take a quick look at that class, just to complete the example.

The ServantData class

The objects created in this class are used as containers to hold the accumulated results of the SMF records as they are processed. When all of the records are processed, each object is called upon to report the results contained in the object.

Create this class the same way you created the plug-in class, right clicking on the project in the navigation pane and selecting New->Class. Remember to place this class in the same package you used for the plug-in.



The Basics

To get started, we declare the class and the four main attributes which correspond to the data items we fetch from each SMF record. We also declare a static attribute used to collect total counts for use in the footer of the results table.

```
package com.ibm.ws390.smf.plugins;

public class ServantData
{
 private String serverName;
 private String STCName;
 private String servant_ASID;
 private int work_counts[];

 private static int total_work_counts[] = new
int[com.ibm.ws390.sm.smfview.PlatformNeutralRequestInfoSection.Type
Max+1];
```

Note that each object instance has an array of counters, one slot for each request type. The static array also has a slot for each type. We assume that TypeOther is the largest type value defined.

The Constructor

The constructor saves away the parameter values in the attribute variables and creates the array of integers used to count the different work types. There is a little sleight of hand to pad out the server name to 8 characters to help keep columns lined up when we print the results.

```
public ServantData(String srv, String stc, String asid)
 {
  serverName = srv;
  if (serverName.length()<8)
  {
   String pad8 = new String("        ");
   String pad = pad8.substring(serverName.length());
   serverName = serverName + pad;
  }
  STCName = stc;
  servant_ASID = asid;
  work_counts = new
int[com.ibm.ws390.sm.smfview.PlatformNeutralRequestInfoSection.Type
Max +1];
 }
```

countWork

As each record is processed we find or create a ServantData object to represent the servant. We have to increment the counter for each request type based on the type of request this SMF record represents. Each object instance has an array with each array slot corresponding to a request type.

We also increment the static counter to keep a total across all servant regions found. Because we have assumed request types are between the Unknown type and Other type, we have to verify the request type we found is in that range to avoid an exception. An error message if the type found is out of bounds would probably be useful to indicate an update to the filter is required.

```
public void countWork(int type)
 {

  if
((type>=com.ibm.ws390.sm.smfview.PlatformNeutralRequestInfoSection.T
ypeUnknown)&&

(type<=com.ibm.ws390.sm.smfview.PlatformNeutralRequestInfoSection.T
ypeMax ))
  {
   ++work_counts[type];
   ++total_work_counts[type];
  }
 }
```

Creating Output

Now that we have all of the data, we need to produce a report. In this section we will go through the three methods used by the filter to produce the report.

Header

This static method is called to generate the header. We return a string which is just column headings that will line up with the data presented by toString().

```
public static String createHeader()
 {
  return new String("Server   JobID
ASID\tUnknown\tIIOP\tHTTP\tHTTPS\tMDBA\tMDBB\tMDBC\tSIP\tSIPS\tMBean
\tOTS\tOther\tTotal");
 }
```

Row Data

The toString() method is called to let each ServantData object present the information contained in the object instance. We begin by concatenating together the server name, the jobname (STCName) and the asid.

Next we iterate through the array of counters. For each counter we concatenate the value to the return string we are building and add the value to a running total. Finally we concatenate the running total of all requests handled by this servant to the end of the result string and return the string to the caller.

```
public String toString()
 {
  String s = new String();
  s = serverName + " " +
       STCName + " " +
      servant_ASID + "\t";

  int total = 0;
  for (int
i=0;i<=com.ibm.ws390.sm.smfview.PlatformNeutralRequestInfoSection.T
ypeMax ;++i)
  {
   s = s + work_counts[i] + "\t";
   total = total + work_counts[i];
  }

  s = s + total;
  return s;
 }
```

Footer

In this static method we print a row of dashes as a divider between the individual servant data and the total values. Next we simply iterate across the static array of counters, concatenating each one to the result string and keeping a running total. At the end we report the total of all requests of all types processed by all of the servants for which we have seen data.

```
public static String createFooter()
 {
 String s = new String();
 int total = 0;
 s = "-------- --------
----\t-------\t----\t----\t-----\t----\t----\t----\t---\t----\t-----
\t---\t-----\t-----\n";
 s = s + "Totals\t\t"+"\t";
 for (int
i=0;i<=com.ibm.ws390.sm.smfview.PlatformNeutralRequestInfoSection.T
ypeMax ;++i)
 {
  s = s + total_work_counts[i] + "\t";
  total = total + total_work_counts[i];
```

```
}
s = s + total;
return s;
}
```

Using Your New Plug-in

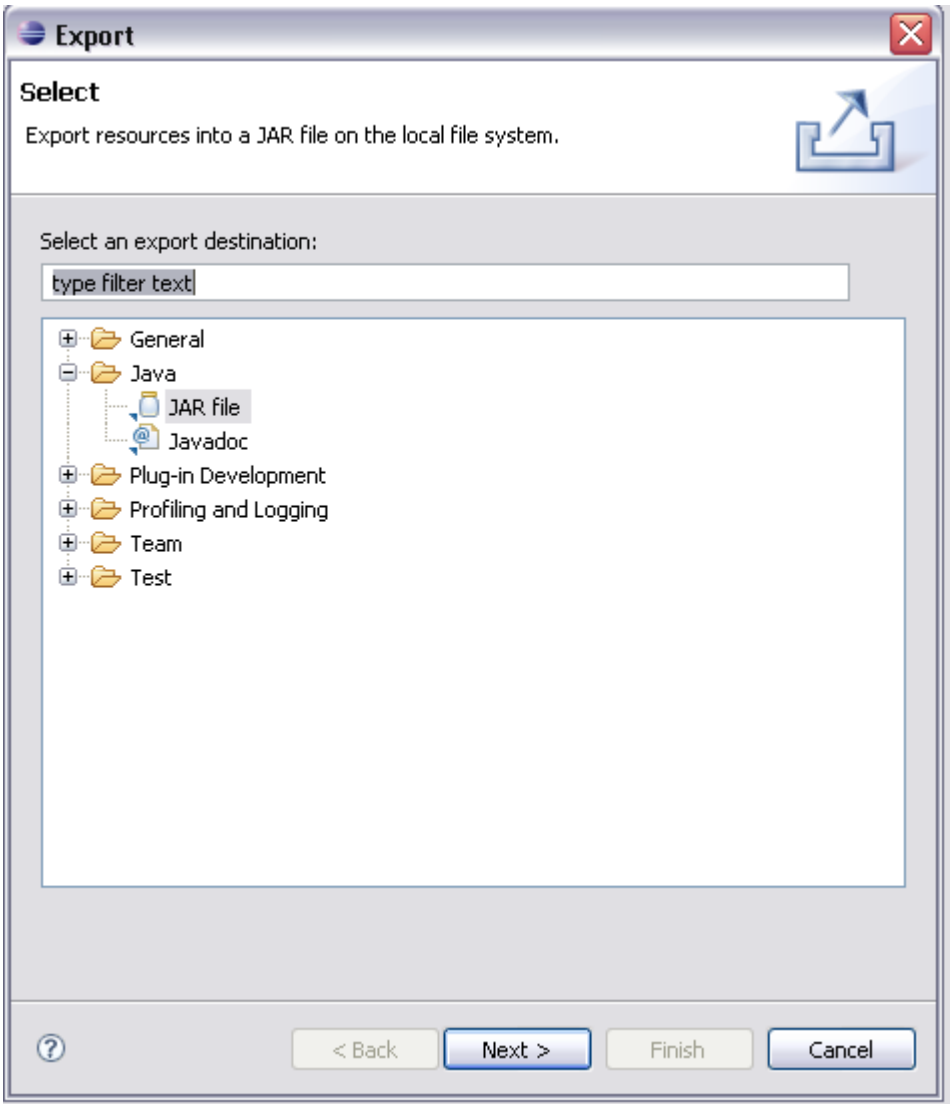
Now that the plug-in is all written, it would be nice to use it.

Export the jar file

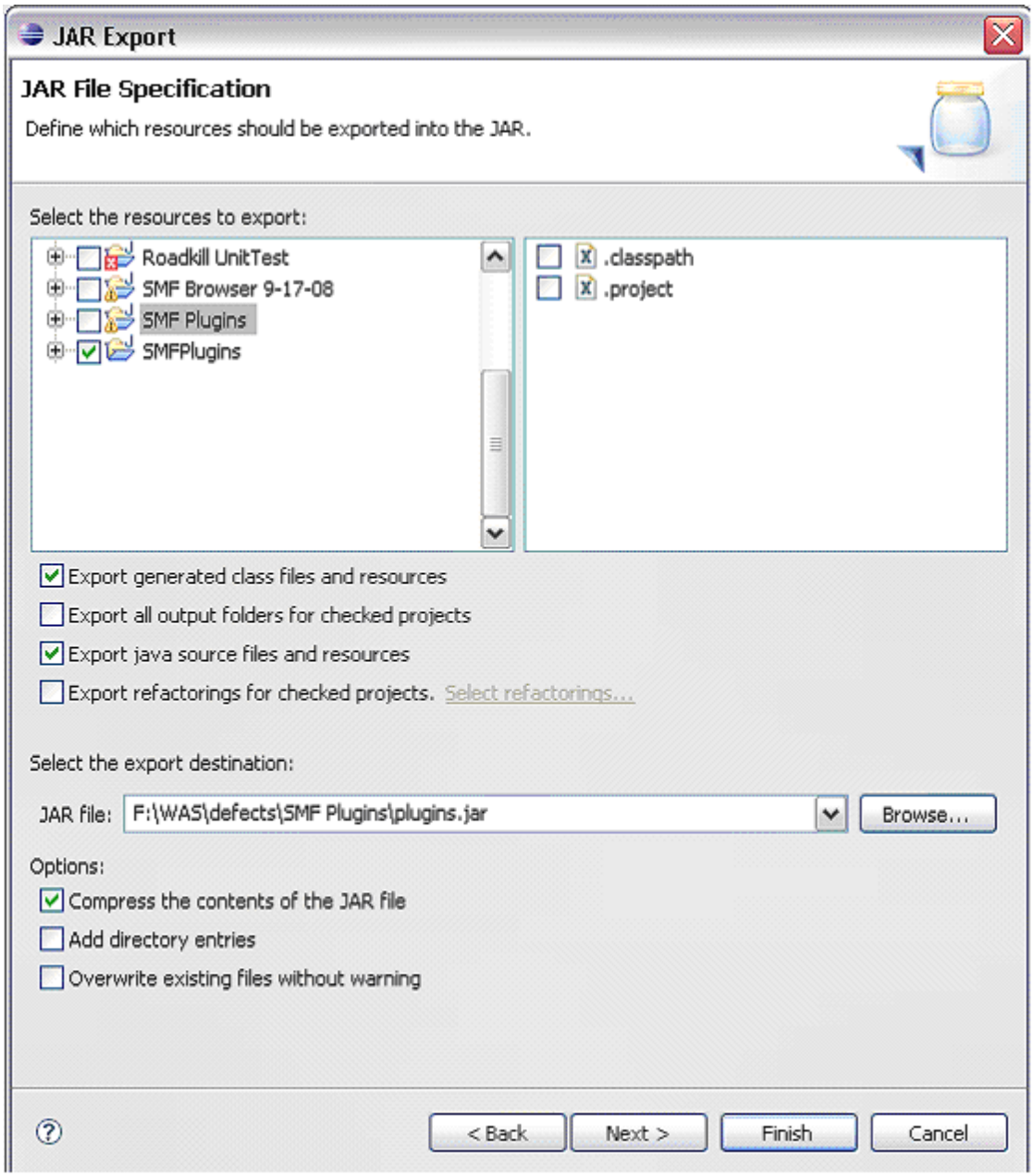
First you need to get the code to your z/OS system. These are the steps:

1. From Eclipse you can right click on the project name in the navigator pane on the left and select ‘Export’.

2. Select “JAR file” as the thing you want to export and click ‘Next’.

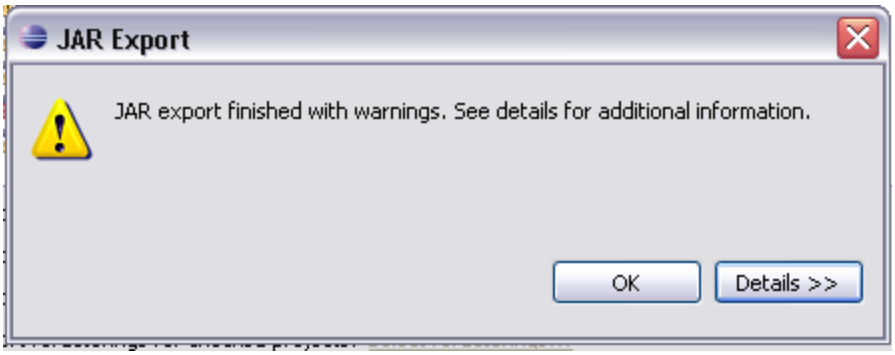


3. Select the project to export and indicate the path and name of the .jar file you want to create.



4. Click ‘Finish’

5. There are probably warnings from the compiler about the HashMap, but it is ok.



When you have the .jar file, use FTP to transfer the file in binary to the USS File System on your z/OS system. Place it in the same directory where you put the bbomsmfv.jar file.

Running the browser

To run the browser with your new plug-in you need to have both your new .jar file (we will assume you called the file plugin.jar) and the bbomsmfv.jar file. These .jar files will need to be on the CLASSPATH for java to use. You might remember the command we used earlier to run the browser with the default plug-in:

```
java –cp bbomsmfv.jar:ibmjzos.jar com.ibm.ws390.sm.smfview.SMF
"INFILE(WAS.SMF.DATA)"
```

We need to update the –cp option to also include the new .jar file. We also need to tell the browser to use our plug-in instead of the default. The new command will look like this (changes in bold):

```
java –cp bbomsmfv.jar:ibmjzos.jar:plugin.jar com.ibm.ws390.sm.smfview.SMF
"INFILE(WAS.SMF.DATA)"
"PLUGIN(com.ibm.ws390.smf.plugins.WorkDistributionSummary,STDOUT)"
```

Notice that the package and class name have to match what you specified in the code. This command will produce output that looks something like this (several columns removed to fit the page):

```
SMF file analysis starts ...
Server   JobID    ASID  Unknown IIOP    HTTP    HTTPS   MBean   OTS     Other
Total
BBOS001  STC00044 003b  0       0       138     71      1       0       47     257
-------- -------- ----  ------- ----    ----    -----   -----   ---     -----
-----
Totals                 0       0       138     71      1       0       47     257

SMF file analysis ended.
```

Other Things

That is all you need to know. Well, perhaps there are a few things we should point out.

Enhancements….

There are a number of minor problems that could be addressed or other issues you might want to consider before using this sample for real purposes.

- We are using the asid of the servant region as a unique key to find the ServantData entry in the HashMap. But servant regions can come and go and asids can be reused. Therefore it is possible to have the same asid represent two different servant regions. The stoken of the servant region would probably be a better choice.
- The report lists the server name of each servant. However, server names are not necessarily unique. It is really the combination of cell, node, and server names which is supposed to be unique. It could be possible that a disaster recovery cell could have exactly the same names as your production cell. If it matters a lot, there are other fields and names in the SMF 120-9 record you can use to try to correctly identify to which server the data applies.
- If a request is received by the server but never gets to the servant region to be dispatched (possibly because a timeout occurred while the request was waiting in the queue) an SMF 120-9 record is still created. This record will be missing certain information such as the asid of the servant region where the request was dispatched. You might want to handle these cases by counting them separately.
- The final report just lists the servant regions found and totals everything up. It would probably be nice to group the servants by server and provide sub-totals for each server.

Problems using an older version of the browser code

The updated browser for WebSphere Application Server Version 7 contained changes to allow you to write your own plug-ins. However, in the course of developing this example and writing this paper we discovered a number of things we could do to make the process a little easier. For example, the Triplet class was not declared as public, which created difficulties fetching fields from the records if your plug-in class was in another package. That is changed now. We also changed the way the default filter initializes to make calling the default initialization process easier. If you have problems getting your plug-in to compile, the reason may be that you are using an older version of bbomsmfv.jar to compile with.

Supporting Other SMF Records

If you would like to get to data kept in other SMF records to enhance whatever report you have decided to generate from the WebSphere records, that is possible too. You just have to implement a java class that knows how to parse the SMF record you are interested in and extract the data that you want.

The browser generates a class name to be used to format each SMF record based on the type and subtype of the record. For example, a Type 120, Subtype 20 record would be formatted by the Java class SMFType120SubType20. Note that the ‘T’ in ‘SubType’ is capitalized for no apparent reason. To handle a Type 72, Subtype 3 record you would create the Java class SMFType72SubType3.

All classes that handle SMF records are expected to be in the com.ibm.ws390.smf.formatters package. Even if your formatter is in a different .jar file (probably best) you need to create this package and put your Java classes in there.

Each class that formats an SMF record needs to extend the com.ibm.sm.smfview.SmfRecord class which is part of the browser.

You should implement a constructor for your class that takes an SmfRecord as an input parameter. This constructor will get control during the parse phase of processing. This is your opportunity to crawl through the record and pull out whatever information you are interested in. You can store this information as attributes of your object.

Your constructor should always first call the super-class' constructor to initialize the base class.

Your class should also implement a dump method. The dump method will be called by the default plugin to print out whatever fields from the SMF record you care to display.

You are, of course, free to provide 'get' methods to make accessing the data you have found in the SMF record easier, or anything else you'd like to provide.

Here is a sample base formatter that does not really do anything:

```
package com.ibm.ws390.smf.formatters;

import com.ibm.ws390.sm.smfview.SmfPrintStream;
import com.ibm.ws390.sm.smfview.SmfRecord;
import com.ibm.ws390.sm.smfview.UnsupportedVersionException;
import java.io.UnsupportedEncodingException;

public class SMFType72SubType30 extends SmfRecord
{
 public SMFType72SubType3(SmfRecord smfRecord)
 throws UnsupportedVersionException, UnsupportedEncodingException
 {
  super(smfRecord);
  // This would be a good place to parse the record!
 }

 public void dump(SmfPrintStream aPrintStream)
 {
  aPrintStream.println("Wish I knew how to format this!");
 }
}
```

Document change history

Check the date in the footer of the document for the version of the document.

July 27, 2010	Initial Version
August 3, 2010	Updated with doc number
June 25,2013	Changed TypeOther to TypeMax (new constant in the browser to make just this sort of thing easier) and to ignore async work (for simplicities sake).
Jul 29,2013	Fixed typos:- Thanks Hutch :-)
January 12,2021	Added note that the browser has moved to github

End of WP101726