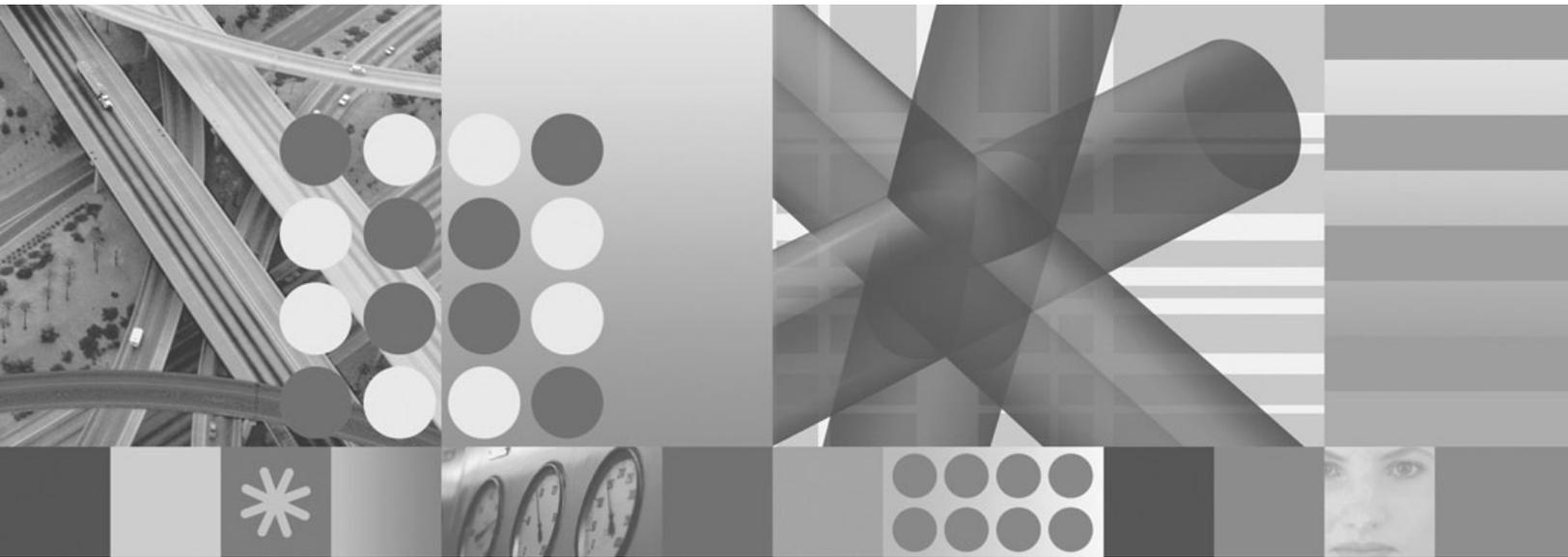




**IBM Tivoli Directory Integrator 6.1.1:
Getting Started Guide**



**IBM Tivoli Directory Integrator 6.1.1:
Getting Started Guide**

Note

Note: Before using this information and the product it supports, read the general information under Appendix B, "Notices," on page 75.

Second Edition (February 2007)

This edition applies to version 6.1.1 of the IBM Tivoli Directory Integrator and to all subsequent releases and modifications until otherwise indicated in new editions.

© Copyright International Business Machines Corporation 2003, 2007. All rights reserved.

US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Preface

This document introduces conceptual information about IBM® Tivoli® Directory Integrator and provides examples to help you get started with the product.

Who should read this book

This book is intended for system administrators and users and anyone interested in learning more about IBM Tivoli Directory Integrator.

Publications

Read the descriptions of the IBM Tivoli Directory Integrator library and the related publications to determine which publications you might find helpful. After you determine the publications you need, refer to the instructions for accessing publications online.

IBM Tivoli Directory Integrator library

The publications in the IBM Tivoli Directory Integrator library are:

IBM Tivoli Directory Integrator 6.1.1: Getting Started

A brief tutorial and introduction to IBM Tivoli Directory Integrator 6.1.1.

IBM Tivoli Directory Integrator 6.1.1: Administrator Guide

Includes complete information for installing the IBM Tivoli Directory Integrator. Includes information about migrating from a previous version of IBM Tivoli Directory Integrator. Includes information about configuring the logging functionality of IBM Tivoli Directory Integrator. Also includes information about the security model underlying the Remote Server API.

IBM Tivoli Directory Integrator 6.1.1: Users Guide

Contains information about using the IBM Tivoli Directory Integrator 6.1.1 tool. Contains instructions for designing solutions using the IBM Tivoli Directory Integrator tool (**ibmditk**) or running the ready-made solutions from the command line (**ibmdisrv**). Also provides information about interfaces, concepts and AssemblyLine/EventHandler creation and management. Includes examples to create interaction and hands-on learning of IBM Tivoli Directory Integrator 6.1.1.

IBM Tivoli Directory Integrator 6.1.1: Reference Guide

Contains detailed information about the individual components of IBM Tivoli Directory Integrator 6.1.1 AssemblyLine (Connectors, EventHandlers, Parsers, Plug-ins, and so forth).

IBM Tivoli Directory Integrator 6.1.1: Problem Determination Guide

Provides information about IBM Tivoli Directory Integrator 6.1.1 tools, resources, and techniques that can aid in the identification and resolution of problems.

IBM Tivoli Directory Integrator 6.1.1: Messages Guide

Provides a list of all informational, warning and error messages associated with the IBM Tivoli Directory Integrator 6.1.1.

IBM Tivoli Directory Integrator 6.1.1: Password Synchronization Plug-ins Guide

Includes complete information for installing and configuring each of the five IBM Password Synchronization Plug-ins: Windows Password

Synchronizer, Sun ONE Directory Server Password Synchronizer, IBM Directory Server Password Synchronizer, Domino Password Synchronizer and Password Synchronizer for UNIX® and Linux®. Also provides configuration instructions for the LDAP Password Store and MQE Password Store.

IBM Tivoli Directory Integrator 6.1.1: Release Notes

Describes new features and late-breaking information about IBM Tivoli Directory Integrator 6.1.1 that did not get included in the documentation.

Related publications

Information related to the IBM Tivoli Directory Integrator is available in the following publications:

- IBM Tivoli Directory Integrator 6.1.1 uses the JNDI client from Sun Microsystems. For information about the JNDI client, refer to the *Java™ Naming and Directory Interface™ 1.2.1 Specification* on the Sun Microsystems Web site at <http://java.sun.com/products/jndi/1.2/javadoc/index.html>.
- The Tivoli Software Library provides a variety of Tivoli publications such as white papers, datasheets, demonstrations, redbooks, and announcement letters. The Tivoli Software Library is available on the Web at: <http://www.ibm.com/software/tivoli/library/>
- The *Tivoli Software Glossary* includes definitions for many of the technical terms related to Tivoli software. The *Tivoli Software Glossary* is available on the World-Wide Web, in English only, at <http://publib.boulder.ibm.com/tividd/glossary/tivoliglossarymst.htm>

Accessing publications online

The publications for this product are available online in Portable Document Format (PDF) or Hypertext Markup Language (HTML) format, or both in the Tivoli software library: <http://www.ibm.com/software/tivoli/library>.

To locate product publications in the library, click the **Product manuals** link on the left side of the Library page. Then, locate and click the name of the product on the Tivoli software information center page.

Information is organized by product and includes READMEs, installation guides, user's guides, administrator's guides, and developer's references as necessary.

Note: To ensure proper printing of PDF publications, select the **Fit to page** check box in the Adobe Acrobat Print window (which is available when you click **File->Print**).

Accessibility

Accessibility features help a user who has a physical disability, such as restricted mobility or limited vision, to use software products successfully. With this product, you can use assistive technologies to hear and navigate the interface. After installation you also can use the keyboard instead of the mouse to operate all features of the graphical user interface.

Contacting IBM Software support

Before contacting IBM Tivoli Software support with a problem, refer to IBM System Management and Tivoli software Web site at:

<http://www.ibm.com/software/sysmgmt/products/support/>

If you need additional help, contact software support by using the methods described in the *IBM Software Support Handbook* at the following Web site:

<http://techsupport.services.ibm.com/guides/handbook.html>

The guide provides the following information:

- Registration and eligibility requirements for receiving support
- Telephone numbers and e-mail addresses, depending on the country in which you are located
- A list of information you must gather before contacting customer support

Contents

Preface	iii
Who should read this book	iii
Publications	iii
IBM Tivoli Directory Integrator library	iii
Related publications	iv
Accessing publications online	iv
Accessibility	iv
Contacting IBM Software support	v
Chapter 1. Introduction	1
About this manual	1
Scripting in JavaScript	2
Installing IBM Tivoli Directory Integrator	2
Installing the tutorial files	2
Chapter 2. Simplify and solve	3
How do you eat an elephant?	3
Integration is communication	3
Architecture	6
AssemblyLines	7
Connectors	9
Parsers	10
Chapter 3. Introducing IBM Tivoli Directory Integrator	11

Rapid integration development	11
Creating a new Config.	12
Creating an AssemblyLine	14
Adding the Input Connector.	19
Mapping Attributes Into The AssemblyLine	27
Adding the Output Connector	36
Running your AssemblyLine	40
Working with Hooks	44
Schema conversion	47
Adding the Join Connector	49
Setting up Link Criteria	52
Event-driven Integration	58
Final thoughts	71

Appendix A. index.html and OtherPage.html	73
index.html.	73
OtherPage.html	73

Appendix B. Notices	75
Trademarks	77

Chapter 1. Introduction

About this manual

This book is a simple introduction to a simple system. Make no mistake, the word **simple** is used here in its most positive and powerful context, because the best way to wrap your mind around a complex problem is to simplify it. Break it down into more manageable pieces, and then master those constituent parts. Divide and conquer. This is a technique that you instinctively use to solve everyday problems, and which is equally relevant for engineering information exchange across an office, an enterprise or the globe.

IBM Tivoli Directory Integrator is designed and built on the premise that integration problems can be broken down into three parts:

- The **systems** involved in the communication
- The **data flows** between these systems
- The **events** which trigger the data flows

With IBM Tivoli Directory Integrator you turn this atomic understanding of the integration problem directly into the solution. You can build your solution incrementally, one flow at a time, with continuous feedback and verification.

This means that integration projects become easier to estimate and plan. Sometimes planning can even be reduced to simply counting and determining the cost of the individual data flows to be implemented. And since you are developing the solution flow-by-flow visually and interactively, you can report (and demonstrate) progress to both project and corporate management at any time.

IBM Tivoli Directory Integrator manages the technicalities of connecting to and interacting with the various data sources that you want to integrate, abstracting away the details of their APIs, transports, protocols and formats. Instead of focusing on data, IBM Tivoli Directory Integrator lifts your view to the information level, enabling you to concentrate on the transformation, filtering and other business logic required to perform each exchange.

IBM Tivoli Directory Integrator enables you to build libraries of components and business logic that can be maintained, extended and reused to address new challenges. Development projects across your organization can all share IBM Tivoli Directory Integrator assets, resulting in independent projects (even point solutions) that immediately fit into a coherent integrated infrastructure.

This approach results in a more rational and predictable use of resources, as you bring your data source and technology experts in at the very start of a project in order to set up your libraries. When in place, these integration assets are available across the network, letting you leverage them to create new solutions and enhance existing ones.

This document gives you an introduction to this approach, as well as information about how to tap into the radical and elegant simplicity of IBM Tivoli Directory Integrator .

Scripting in JavaScript

IBM Tivoli Directory Integrator provides an elegant and intuitive point-and-shoot environment for rapidly building the framework of your integration solution. However, you might soon want to add more advanced data manipulation and transformation logic, as well as business rules for filtering your data and controlling the behavior of your data flows. All of this is done by writing script in your solution.

Scripting is done in JavaScript, and TDI includes the IBM jsEngine to provide a fast, reliable scripting environment.

Note: It is no longer possible to choose the scripting language for a component or AssemblyLine. Scripting is always done in JavaScript.

For more information about scripting in IBM Tivoli Directory Integrator, see the *IBM Tivoli Directory Integrator 6.1.1: Users Guide*.

Installing IBM Tivoli Directory Integrator

IBM Tivoli Directory Integrator is light-footed, rapidly deployed integration middleware. Unlike traditional middleware, IBM Tivoli Directory Integrator installs in minutes and you can begin building, testing and deploying solutions immediately. The system runs on a wide variety of platforms, including Windows® and a number of UNIX and Linux versions.

For more information about installing the IBM Tivoli Directory Integrator, please see "IBM Tivoli Directory Integrator installation instructions" in the *IBM Tivoli Directory Integrator 6.1.1: Administrator Guide*.

Installing the tutorial files

To work through the examples in this manual, you need to create a Config File to work with. See "Creating a new Config" on page 12 for directions on how to create a Config File.

Supporting data files to use with the Config File are located in the *root_directory/examples* directory in the installation directory.

root_directory indicates the directory where IBM Tivoli Directory Integrator is installed¹.

1. During the install process you are asked to specify the location of your solutions directory. This is where you will keep your own work, and will typically be a sub-directory under your *home* area. However, if you are upgrading from a version older than 6.0, this will typically set to the installation *root_directory*.

Chapter 2. Simplify and solve

How do you eat an elephant?

The answer is, one bite at a time. This is also the best approach for digesting large integration and systems deployment projects.

The key to success is to reduce complexity by breaking the problem up into smaller, manageable pieces. This means starting with a portion of the overall solution, preferably one that can be completed in a week or two. Ideally, this is a piece that can be independently put into production. That way, it's already providing return on investment while you tackle the rest of the solution.

When you have isolated the piece you are going to work with, simplify it further by focusing in on the basic units of communication (the data flows themselves). You are now poised to start implementing them.

Integration development is done with IBM Tivoli Directory Integrator through a series of try-test-refine cycles, making the process an iterative, even exploratory one. This not only helps you to discover more about your own installation, but also lets you grow your integration solution as your understanding of the problem set and its impact on your infrastructure grows.

A great way to get a good mental picture of the problem at hand is to make a picture of it. Using a pencil and a piece of paper, sketch out a flow diagram that maps out the solution in broad strokes. This exercise not only helps you to visualize the scope of the task, it serves as a blueprint for implementing the task in IBM Tivoli Directory Integrator.

Integration is communication

Integration problems are all about communication, and as such can typically be broken down into three basic parts:

- The systems and devices that communicate
- The flows of data between these systems
- The events that trigger the data flows

These constituent elements of a communications scenario can be described as follows:

Data Sources

These are the data repositories, systems and devices that talk to each other. For example:

- The Enterprise Directory you're implementing or trying to maintain
- Your CRM application
- The office phone system
- The Access database with the list of company equipment and to whom the equipment has been issued

Data sources represent a wide variety of systems and repositories, such as databases (for example, DB2®, Oracle, SQL Server), directories (for example, iPlanet, IBM Directory Server, Domino™, eDirectory and Active Directory), directory services (Exchange), files (for example, XML, LDIF or

SOAP documents), specially formatted e-mail, or any number of interfacing mechanisms that internal systems and external business partners use to communicate with your information assets and services.

Data Flows

These are the threads of the communications and their content, and are usually drawn as arrows which point in the direction of data movement.

Each data flow represents a communication between two or more systems.

However, for a conversation to be meaningful to all participants, everyone involved must understand what is being communicated. You can probably expect the data sources to represent their data content in different ways. One system might represent a telephone number as textual information, including the dashes and parentheses used to make the number easier to read. Another system might store the telephone numbers as numerical data.

If these two systems are to communicate this data, then the information must be translated during the conversation. Furthermore, the information in one source might not be complete, and might need to be augmented with attributes from other data sources. Furthermore, only parts of the data in the flow might be relevant to receiving systems.

Therefore a data flow must also include the mapping, filtering and transformation of information, shifting its context from input sources to that of the destination systems.

Events

Events can be described as the circumstances dictate when one set of data sources communicates with another. One example is whenever an employee is added to, updated within or deleted from the HR system. Another example is when the access control system detects a keycard being used in a restricted area. An event can also be based on a calendar or a clock-based timer, for example, starting communications at every 10 minutes, or at 12:00 midnight on Sundays. It can also be a manually initiated one-off event, such as populating a directory or washing the data in a system.

Events are usually tied to a data source, and are related to the data flows that are triggered when the specified set of circumstances arise.

These elements are all handled by the IBM Tivoli Directory Integrator component called a *Connector*². Connectors are components that connect to and access data in a data source. For example, you can use a JDBC Connector to read and write to an SQL database, or an LDAP Connector to access directory information. Connectors can also be set up to handle events from a data source; like changes in a directory or database, mail arriving in a mailbox and messages appearing in a message queue.

Some types of data sources do not store data as structured objects (records, entries, and so forth). Instead, data is represented as a byte stream. Two examples are data over IP and flat files. That's where a second type of component called a *Parser* is necessary. By attaching a Parser to a Connector that is working with an

2. In addition to Connectors, IBM Tivoli Directory Integrator provides a number of other components, like Functions and Scripts. However, only Connectors, Parsers and EventHandlers will be covered in this text.

unstructured data source, byte streams are converted structured information during input operations, and structured information into byte streams on output.

Before diving deeper into how these components work, lets start working on an example. The first order of business is to get an overview of the data flows you want to implement. As mentioned previously, a good way to do this is to create a flow diagram³.

In this example, an output data source (DS3) will receive data from an input data source (DS1). Along the way, the data flow will *aggregate* from a second input data source (DS2) — also called a *join* operation. In IBM Tivoli Directory Integrator, the implementation of such a data flow is referred to as an *AssemblyLine*.

It's important to understand that each AssemblyLine implements a single uni-directional data flow. If you want to do bi-directional synchronization between two or more data sources, then you should use a separate AssemblyLine for handling the flow in each direction. The reason for this is that the form and content of the data, as well as the operations carried out on it, most likely are different for each direction⁴.

Note: IBM Tivoli Directory Integrator provides everything needed to create request-response information solutions such as Web Services. This type of bi-directional data flow is supported by built-in AssemblyLine behavior.

Although there are no limits to the number of Connectors that an AssemblyLine can contain, the AssemblyLines should contain as few Connectors as possible (for example, one per data source participating in the flow). This is a best practice: keeping AssemblyLines as short and simple as possible. At the same time, you need to include enough components and script logic to make the AssemblyLine as autonomous as possible. The reasoning behind this is to make the AssemblyLine easy to understand and maintain. It also results in simpler, faster and more scalable solutions.

The IBM Tivoli Directory Integrator philosophy is about dealing with the flows one at a time, as well as building your solution incrementally. So instead of attacking the entire problem at once, let's start with the just a data flow going from DS1 to DS3.

How data is organized can differ greatly from system to system:

- Databases typically store information in *rows* typically with a fixed number of columns.
- Directories, on the other hand, work with variable objects called *entries*.
- Message Queues deliver information as *messages*.
- Other systems represent data as records, objects, byte streams or key-value pairs.

IBM Tivoli Directory Integrator simplifies this issue by normalizing the way data is represented inside your AssemblyLine. This means that regardless of where the

3. There are many diagramming conventions and styles available to choose from, but the actual shape and type of symbols is less important than your understanding of the problem. Use boxes or balls or bubbles or whatever you're comfortable with, but be consistent and be sure to label everything clearly and legibly. That way, when you look at your diagram in a couple of months (or when someone else does), it still makes sense.

4. This is a best practice with IBM Tivoli Directory Integrator, and is supported by built-in behaviors of the system. However, since everything can be changed and controlled via JavaScript code, you can override built-in flow control to implement pretty much what you want.

information comes from, or how it is stored in your data sources, it is handled inside the system in a canonical format: Java objects. Each component knows how to marshal data between its data source's native types and their corresponding Java representation. This makes working with the data a lot easier, since you don't have to worry about type conflicts when doing data comparisons and computations.

Getting back to the exercise, your output data source (DS3) has a schema of five attributes: **First**, **Last**, **FullName**, **Title** and **Mail**. The input data source (DS1) provides you with only three attributes: **First**, **Last** and **Title**. You need to complete the design of your data flow by deciding how the attributes provided by DS1 are mapped (and possibly transformed) to provide those required by DS3:

```
DS3.First      =DS1.First
DS3.Last       =DS1.Last
DS3.FullName   =DS1.First+" "+DS1.Last
DS3.Title      =DS1.Title
DS3.Mail       =<compute from DS1.FullName>
```

The above specification indicates that while three of the attributes needed by DS3 (**First**, **Last** and **Title**) can be directly mapped from DS1, the two remaining ones will need to be computed. This mapping specification will be implemented directly in your IBM Tivoli Directory Integrator solution, as you will see soon.

To keep this example simple, a comma-separated text file will serve as DS1. It will contain the fields **First**, **Last** and **Title**. The output data source (DS3) will be an XML document that your solution will create.

Now that you have a good representation of the solution and understand how IBM Tivoli Directory Integrator deals with data, let's take a look at how data flows are handled by the system.

Architecture

The architecture of IBM Tivoli Directory Integrator is divided into two parts:

- The **kernel**, where most of the system's functionality is provided, and which you leverage to quickly build the framework of your solution.
- The **components**, which abstract away the technical details of the data systems, platforms and formats that you want to work with. IBM Tivoli Directory Integrator provides you with a number of components types, although this manual will be focusing on two: **Connectors** and **Parsers**. Connectors are the main type of component, and they tie your data flow to the outside world. Parsers are used to translate byte streams into structured data, or vice versa, allowing your Connectors to access files and message queues, as well as communicate using IP protocols.

IBM Tivoli Directory Integrator has a rich component library for you to work with, each of them specialized to handle a particular API, protocol or format. In addition to this data source specific "intelligence", components are wrapped with kernel functionality that harmonizes their behavior and does much of the "heavy-lifting" in your solution. As a result, components become not only easy to configure and use, but also interchangeable. And the components themselves can remain relatively simple in design and implementation, making it easy to extend them and build new ones — either by writing them in Java, or directly in the IBM Tivoli Directory Integrator development environment (called the Config Editor, or CE for short) using JavaScript.

As you will see, this **kernel/component** design philosophy is a core concept in IBM Tivoli Directory Integrator. While specialized component technology gives you access to a broad range of systems and devices, the generic functionality of the kernel lets you rapidly build the framework of your solutions by selecting relevant components and clicking them into place. Furthermore, components are interchangeable and can be swapped out without affecting the customized logic and configured behavior of your data flows. Your integration solutions become agile and extensible, making them less vulnerable to changes in the underlying infrastructure.

Another central architectural concept in IBM Tivoli Directory Integrator is how it collects and stores information in a powerful and flexible data container called an **Entry**. Entries are objects that can be thought of as "Java buckets" used to carry information about a single data entity, like a database row, directory entry or MQ message.

Entries can hold any number of **Attributes**, which in turn store the actual data *values* themselves.

As an example, consider an Entry representing a row in a database table called "Cars". This Entry would have an Attribute for each column in the table, like "Make", "Model", "Manufactured" and "Owners". These attributes would in turn contain the data values stored in the row: for example, "Volkswagen", "Caravelle", 1997 and *Bill Sanderman*. As mentioned above, these Attribute values are stored as Java objects, converted from native database types by the Connector you used to read the data. So while the Attributes above called "Make", "Model" and "Owners" contained string values (`java.lang.String`), "Manufactured" is represented as a date (`java.util.Date`).

Some data sources (like directories) support multiple values for a single attribute. IBM Tivoli Directory Integrator Attributes can also hold any number of values⁵. In the example outlined above, the car in question could have a history of owners, represented by multiple string values stored in the "Owners" Attribute.

As stated above, this Entry->Attribute(s)->value(s) concept is core to understanding and using with IBM Tivoli Directory Integrator. Whenever you deal with data inside your AssemblyLine, you will be working with Entries and Attributes (and their values).

AssemblyLines

The data flow arrows in your solution diagram translate in IBM Tivoli Directory Integrator to AssemblyLines. The AssemblyLine itself is an ordered list of components that forms a single path of data transfer and transformation in your solution. Built-in behavior provided by the kernel ties the components together and passes data from one to the next. As you probably guessed, this is where the AssemblyLine name comes from: real-world industrial assembly lines.

Real-world assembly lines are made up of a number of specialized machines that differ in both function and construction, but have one significant attribute in common: they can be linked together to form a continuous path from input sources to output targets.

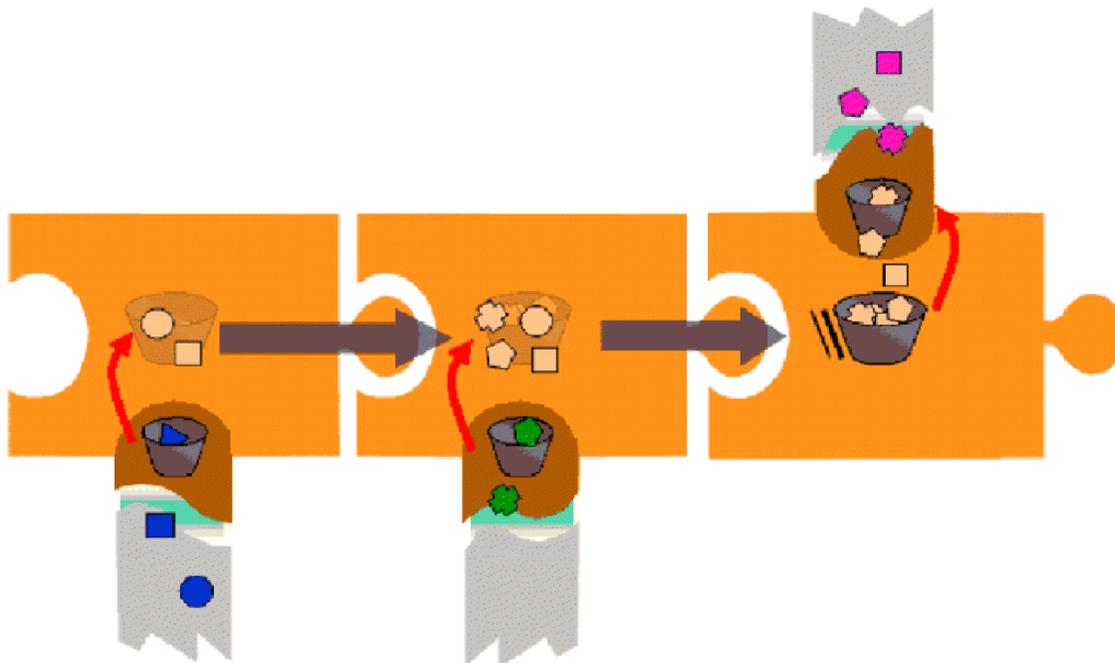
5. Furthermore, since a value can be any type of Java object, even another *Entry*, you can work with hierarchical data sets.

An assembly line generally has one or more input units designed to feed the production process with raw materials: for example, fish fillets, cola syrup, car parts, and so forth. These ingredients are processed and refined, carried from one unit to the next by some transport mechanism like a conveyor belt or pipes. Sometimes by-products are extracted from the assembly line along the way. At the end of the line, finished goods are delivered for to output bins for stacking, storage or distribution.

If a production crew gets the order to produce something else, they break the line down, keeping the machines that are still relevant to the new order. New units are connected in the right places, the line is adjusted and production starts again. IBM Tivoli Directory Integrator AssemblyLines work in much the same way.

IBM Tivoli Directory Integrator AssemblyLines are fed information from various input units (Connectors), perform operations on this data and then convey the finished product to target systems through output units (also Connectors). IBM Tivoli Directory Integrator AssemblyLines process on one item at a time: for example, one database row, directory entry, MQ message, and so forth. Just like real-world assembly lines, IBM Tivoli Directory Integrator AssemblyLines need some "conveyor belt" mechanism for moving data down the flow.

Data transport within the AssemblyLine is done by storing data attributes read from the connected input sources into an Entry object (the "Java bucket" mentioned previously) called the **work** Entry object. The **work** object is passed between AssemblyLine components which in turn perform work on the information it contains — for example, joining in additional data, verifying content, computing new attributes and values, as well as changing existing ones — until the data is ready for delivery to one or more target systems.



It's said that a picture is worth a thousand words, and the above diagram is no exception. Start with the three puzzle pieces which represent Connectors linked together to form an AssemblyLine. The darker "stem" of each puzzle piece highlights the data source specific part of the Connector; i.e. the *component* part

that is connected to some system or device, and which has the "intelligence" to work with a particular API or protocol. The rest of each puzzle piece is the generic functionality of the Connector provided by the kernel. This AssemblyLine "wrapper" makes components work in a similar and predictable fashion. It enables AssemblyLine components to be linked together, as well as providing built-in behaviors and control points for customization. As you can see, every AssemblyLine component reflects the kernel/component architecture of IBM Tivoli Directory Integrator.

In addition to the **work** object used by the AssemblyLine to move data down the flow, the diagram also shows an additional "Java bucket" nestled in each of the Connectors. These local storage objects are used to cache data during read and write operations. A Connector's local Entry object is call its **conn** object, and exists only within the context of the Connector. When a Connector reads information, it converts the data to Java objects and stores it in the local **conn** object. During output, the Connector takes the contents of its **conn**, converts this data to native types and sends to the target system.

However, since each **conn** object is only accessible by its Connector, an additional mechanism is needed to move data from these localized caches to the shared **work** object after Connector input — and the other direction for output Connectors. The above diagram shows arcing arrows that illustrate this movement of Attributes between the Connectors' local **conn** Entries and the AssemblyLine's **work** object. The process is called *Attribute Mapping* and will be detailed later. Suffice to say that Attribute Maps are your instructions to a Connector on which Attributes are to be brought into the AssemblyLine during input, or included in output operations.

Keep this image handy, as it will help you to understand exactly how the AssemblyLine you are going to build actually works.

Connectors

Connectors are the puzzle pieces that you click together to build your AssemblyLine. Each one is designed to tie a specific data source to your data flow.

Each time you select one of these puzzle pieces and add it to an AssemblyLine, you must do the following:

1. Choose the type of Connector to use and configure it.
2. Assign the Connector its *role* in the data flow so that the built-in automated behavior of the AssemblyLine can power the Connector for you. This is called the Connector **Mode** setting, and is determines whether you are want:
 - an input Connector iterating through or looking up information in its source;
 - an output Connector, inserting, updating or deleting data in the connected system or device.

You can change both the type and mode of a Connector at any time in order to meet changes in your infrastructure, or in the goals of your solution. If you've planned for this eventuality, then the rest of the AssemblyLine is not impacted. That's why it's important to treat each Connector as a *black box* that either delivers data into the mix, or extracts some to send to an output target. The more independent each Connector is, the easier your solution is to augment and maintain.

By making your Connectors as autonomous as possible, you can also readily transfer them to your Connector Library and reuse them to create new solutions

faster — even sharing them with colleagues. Using the IBM Tivoli Directory Integrator library feature also makes maintaining and enhancing your Connectors easier. Whenever you update the Connector *template* in your library, all AssemblyLines derived from it inherit these changes and enhancements.

IBM Tivoli Directory Integrator gives you a rich selection of Connectors to choose from: such as LDAP, JDBC, Microsoft® NT4 Domain, Lotus® Notes® and POP3/IMAP to name a few. And if you can't find the one you are looking for, you can extend an existing Connector by overriding any or all of its functions using JavaScript™. You can even create your own with either JavaScript, or with a traditional development language like Java or C/C++.

IBM Tivoli Directory Integrator also supports most transport protocols and mechanisms, such as TCP, SNMP, FTP, HTTP and JMS (MQ), allowing you to take advantage of the technologies available in your infrastructure.

Parsers

Even unstructured data (such as text files or data coming over an IP port) is handled quickly and simply with IBM Tivoli Directory Integrator by passing the byte stream through one or more Parsers. Parsers are another type of IBM Tivoli Directory Integrator component, and the system includes a variety of Parsers, such as LDIF, DSML, XML, CSV and Fixed-length field. And just like Connectors, you can extend and modify these, as well as create your own.

Continuing with the example from page 6, the next step is to identify the data sources. Since the input data source is a text file in comma-separated value format, you use the File System Connector paired up with the CSV Parser. Use a File System Connector for output as well, but this time choose the XML Parser in order to format the file as an XML document.

Note: The examples in this manual have been created on a UNIX platform, and use the UNIX path name conventions. In order for your solution to be platform independent, use the forward slash (/) instead of the backslash character (\) in your path names, for example, examples/Tutorial/Tutorial1.cfg. This works on both Windows and UNIX/Linux platforms.

Before you continue, you must have an input file. You can find an example of such a file in the examples/Tutorial sub-directory in the directory where the IBM Tivoli Directory Integrator was installed, or you can create your own with a text editor. The included sample data looks like the following:

```
First;Last;Title
Bill;Sanderman;Chief Scientist
Mick;Kamerun;CEO
Jill;Vox;CTO
Roger
Gregory;Highpeak;VP Product Development
Ernie;Hazzle;Chief Evangelist
Peter;Belamy;Business Support Manager
```

This file should be called People.csv. Once it is in place, you are ready to build your solution using IBM Tivoli Directory Integrator.

Chapter 3. Introducing IBM Tivoli Directory Integrator

Rapid integration development

IBM Tivoli Directory Integrator is actually two programs:

Config Editor (or *CE* for short)

This program gives you a graphical interface to create, test and debug your integration solutions. The CE is an Integrated Development Environment (IDE) used to create a configuration file that describes your solution, and is powered by the runtime Server. This configuration is called a *Config*, hence the name Config Editor. The CE executable file is **ibmditk**.

Server

By reading and interpreting the Config you created with the CE, the Server is able to power your integration solution. The Server program file is **ibmdisrv**, and you will typically deploy your solution by using a single Server instance (although you can have as many Servers as you want, running separately or in concert).

Start the Config Editor. After a moment you are presented with the Main Screen. At this point you can create your new Config.

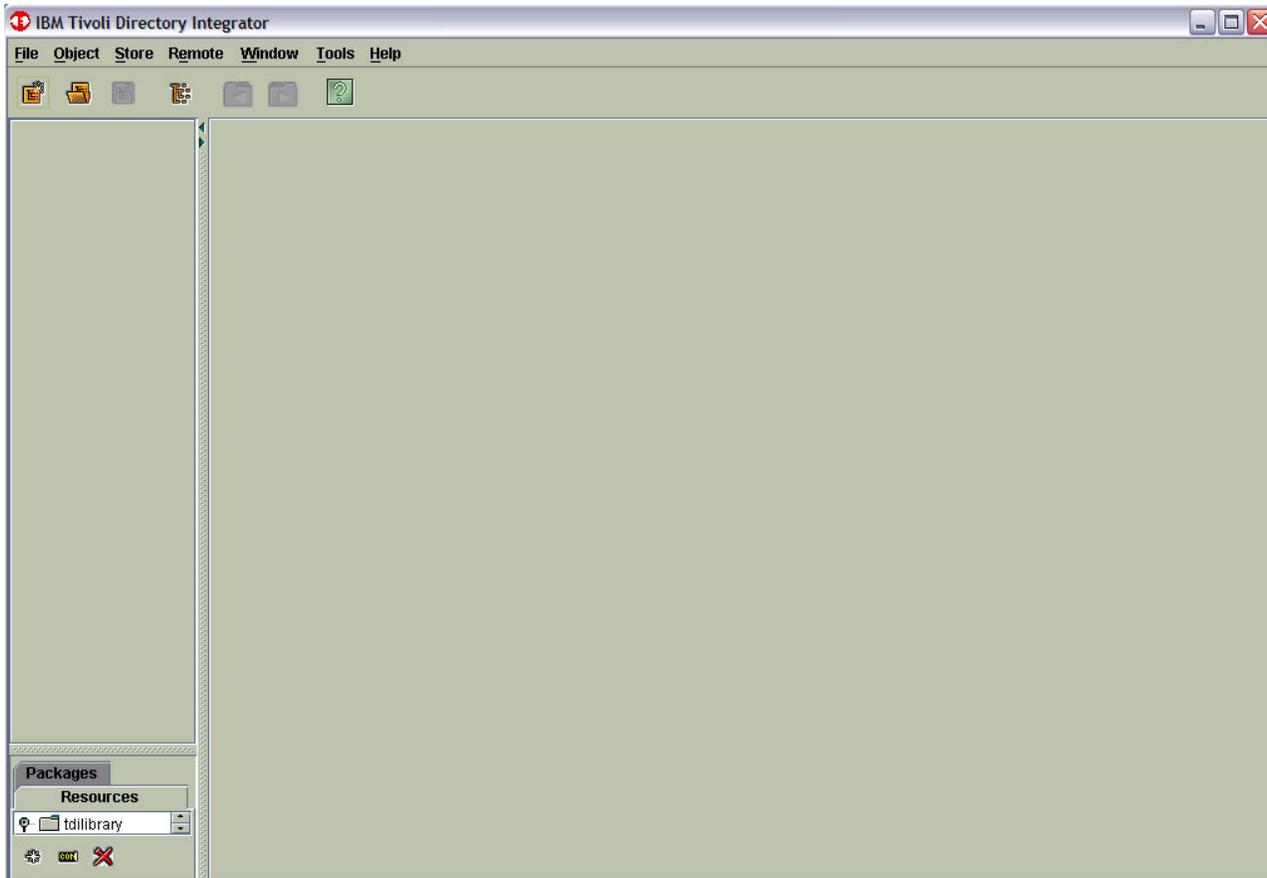
Note: If the screen you see is different from the screenshots in this manual, your system might have a different display setting. Do the following to change the display setting:

1. Click **File->Edit Preferences**.
2. Click the **Appearance** tab.
3. Click the **Look & Feel** tab.

The IBM Tivoli Directory Integrator window is resizeable, as well as the various details panes within the program window. The **File->Edit Preferences** selection opens a dialog where you can set a number of other user interface parameters, such as whether you want a tabbed Browser or Details display, the main button toolbar visible or not, or if you want IBM Tivoli Directory Integrator to show the Status Bar at the bottom of the window.

At the top of the screen is the Main Menu and the Main Toolbar.

Screen capture filename: GettingStarted-11.eps



The Main Toolbar provides commands for creating new Configs, opening existing ones and saving your current work, as well as buttons for window navigation and Help. These same commands are also available under the **File** and **Window** menus, where you also find the **Save As** selection for saving your configuration to a new filename⁶.

Creating a new Config

IBM Tivoli Directory Integrator Configs are stored as XML documents. They are created and maintained in the Config Editor and deployed with the Server. Each Config contains the AssemblyLines that a Server runs, as well as the IBM Tivoli Directory Integrator components that make up these lines.

Note: Configs can also be spread over several files and stored at several locations. IBM Tivoli Directory Integrator assembles its configuration dynamically at startup, using included URLs and filepaths that you have specified. This means that you can create and maintain corporate settings and components that can be shared by many developers. The Config Editor allows you to work with several Configs at once, dragging and dropping components between them. You can even authenticate and connect to a running Server (locally, or on a remote machine) and work running Configs.

6. There are also buttons for creating and opening Configs on a remote Server. However, this functionality will not be covered in this manual

When you start the system for the first time, you are presented with the empty screen as pictured previously. Click the **Create a New Configuration** button, or use the **File->New** menu selection to create a Config called **Tutorial1** (note that the .xml extension is **not** added for you). This file should be saved in the `examples/Tutorial` directory.

Screen capture filename: GettingStarted-15.eps



Note: path names can be written as relative to the solution directory that you specified when installing IBM Tivoli Directory Integrator.

Once you have created or opened a Config, the CE display changes, showing you the contents of this solution description. The tree-view at the left side of the screen is called the **Config Browser**, and it presents you with a set of folders that contain various aspects of your solution.

The topmost folder is called **Config** and contains three items: **AutoStart**, **Logging** and **Tombstones**. AutoStart is used to specify which AssemblyLines (or EventHandlers) are to be launched when the Server starts up. You include items for automatic startup by dragging them from the Config Browser into the AutoStart window. The second item, Logging, is used to specify how logs are to be handled for this Config⁷. The third item, Tombstones, allows you to create tombstones for the selected configuration, AssemblyLines and EventHandlers.

The second folder called **AssemblyLines** will hold the AssemblyLines you create. Just below it are six folders which make up your *Component Library* (one folder for each type of component). This enables you to set up standard Connectors (for example, LDAP, JDBC, Notes, and so forth) with configuration parameters and behavior, and then use and reuse them to create new solutions.

Without going into the Config Editor interface in great detail, here is the general layout of the screen:

- Most of the CE window is organized into *panes*. The Config Browser appears at the left, and you can hide and show this pane by using the **Toggle Configuration Tree View** button in the main Toolbar, or by clicking on the arrows at the top of the vertical divider bar.
- To the right of this pane is a Details area which changes to show the details of items that you select in the Config Browser. The Details pane can show multiple detailed views, opening a new view each time you select another item in the Config Browser. These panes can be accessed using the **Window** menu, the Next and Previous tab/window buttons in the Main Toolbar, or by clicking on the tabs at the top of each Details view⁸.
- The Detail view adapts to show the particulars items selected.
- Some Detail panes contain Element Lists, and you can change the space allocated for any column by moving the mouse cursor over the boundary

7. Note that you can also specify Logging for individual AssemblyLines and EventHandlers which is applied *in addition* to any specification done at the Config level.

8. This depends on the **View Type** setting in the **Appearance** tab under **File->Preferences**. Note that the screenshots in this manual are of a system with the **View Type** set to **Tabbed**

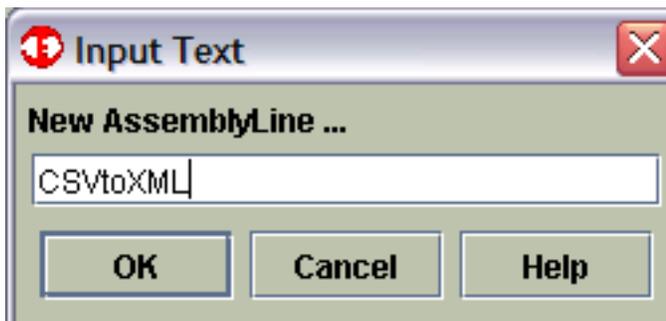
between column titles (the cursor indicates that you can do so by changing shape), and then clicking and dragging it to the desired size.

- At the top of each Element List is a row of buttons providing the set of operations that are available for that type of object (except for the AL Component List and Config Browser tree-view, where the button bar appears at the bottom). The list of available operations varies for the different Element Lists, but the general behavior is the same: you select one or more item in the list and then click the button to manipulate it; with the exception of any **Add** button, which does not require you to make a selection first.
- You can select several list items at once by pressing Shift or Ctrl.
- You can drag items from the Config Browser into your AssemblyLines, or between open Configs.

Creating an AssemblyLine

You will now create your first AssemblyLine. Right-click on the AssemblyLine Folder and select **New AssemblyLine**. Name this AssemblyLine **CSVtoXML**.

Screen capture filename: GettingStarted-18.eps



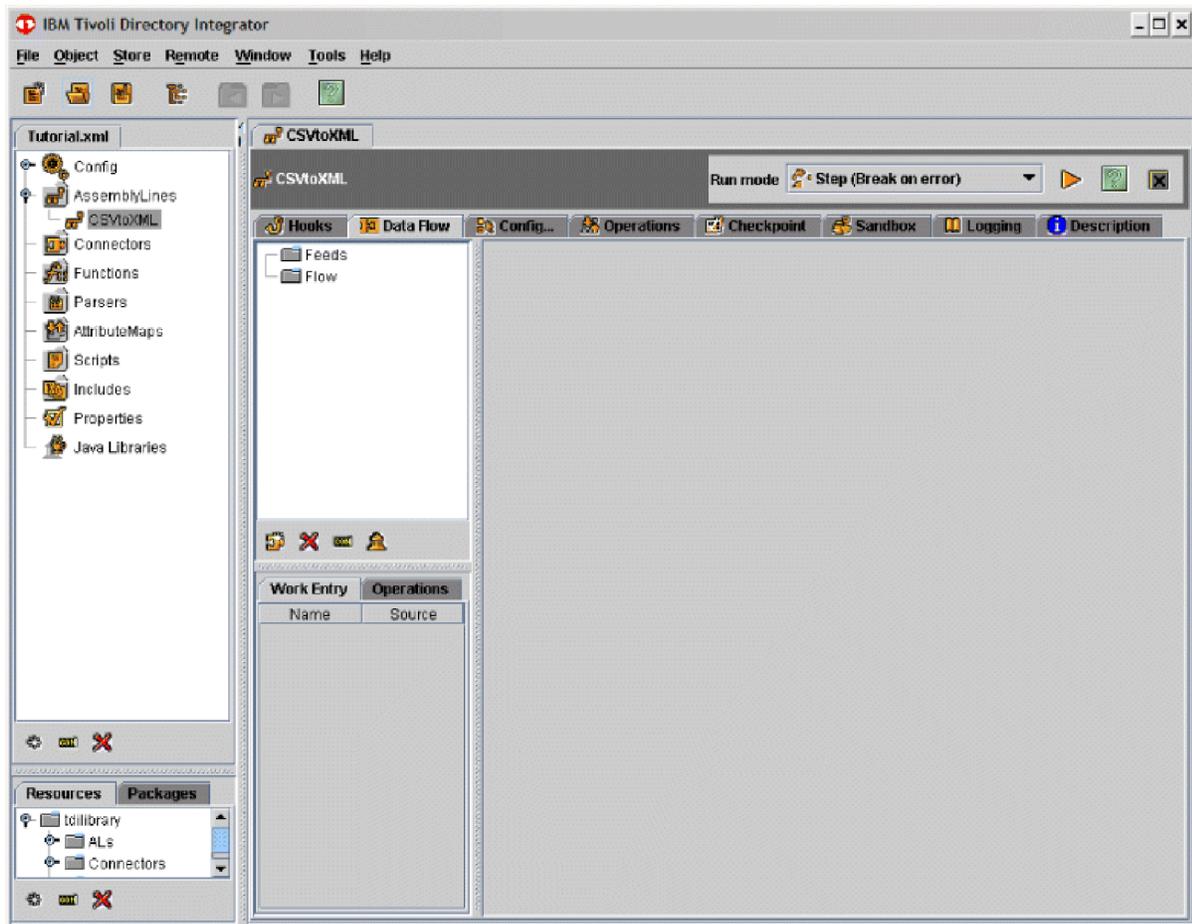
You can name an AssemblyLine whatever you want, but it is important to use a naming convention that helps to document your solution.

Note: Use of special characters and spaces in naming AssemblyLines or IBM Tivoli Directory Integrator components (such as Connectors and EventHandlers) is not a good idea, as it might cause problems later when you want to start IBM Tivoli Directory Integrator Server from a command prompt to run your solution. Furthermore, AssemblyLine components are automatically registered as script variables, enabling you to directly manipulate and re-configure them at runtime. So a good rule of thumb is to use legal variable names on all Config items: start them with a letter, followed by letters, digits and the underscore (_) symbol.

IBM Tivoli Directory Integrator now shows you to the AssemblyLine screen. Notice that this new screen fills the previously-empty Details pane.

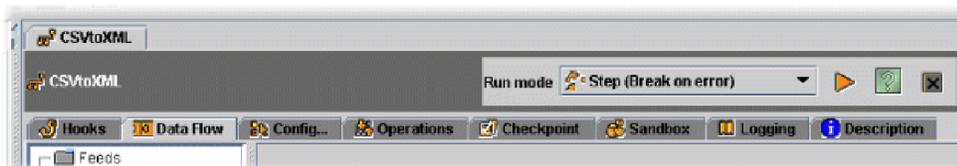
Before adding the Connectors, take a quick look at the layout of the AssemblyLine screen:

Screen capture filename: GettingStarted-19.eps



At the top of the Detail pane is a row of tabs, each associated with an open item from the Config Browser (in this case, your new *CSVtoXML* AssemblyLine). In the title area of each tab a is button row that includes a **Close** button at the far right for closing this details screen.

Screen capture filename: GettingStarted-20.eps



Note: ToolTips appear if you let the mouse hover over buttons or labels in the Config Editor.

Along with **Close**, the button row also includes a selection and three other buttons:

Run Mode (selection)

Lets you select the **Run Mode** for this AssemblyLine:

Step (break on error)

This is the default **Run Mode**. **Step (break on error)** causes the AssemblyLine to execute normally until an error occurs. When an error

occurs, the **AssemblyLine Stepper/Debugger** window appears, allowing you to examine the data and to debug the exception or error.

Step (paused)

In this mode, the AssemblyLine initializes and then pauses in the **Stepper/Debugger**. You can now use the **Stepper/Debugger** to step through AL execution, to set breakpoints, and to watch or modify data during processing.

Standard (run to completion)

This is the fastest run mode and does not invoke the AssemblyLine **Stepper/Debugger**.

Standard (record)

This mode allows you to record input data from Connectors during execution.

Standard (playback)

This mode executes a recorded AssemblyLine run.

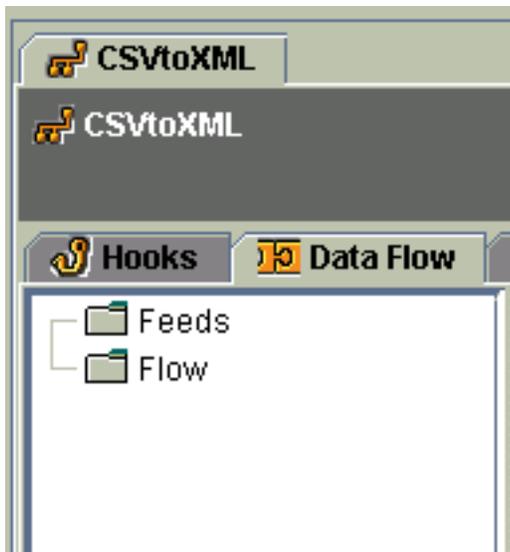
Step (playback)

This mode lets you debug a recorded AssemblyLine run.

Help Opens the Help system.

The white box on the left side of the AssemblyLine Details screen is the AL Component List. This is where new components (for example, Connectors) appear as they are added to your AssemblyLine.

Screen capture filename: GettingStarted-22b.eps



As you can see, the AL Component List is divided into two main sections: *Feeds* and *Flow*. There is also an additional section called *Response* which you can ignore for now, as it is only used in special situations. This division reflects the requirements and behavior of the AssemblyLine, as well as making the AssemblyLine more legible. Start by looking at the *Feeds* section.

In order for your AssemblyLine to do any work, it must be fed with data in the form of Entry objects. These can either come from Connectors in the *Feeds* section, or passed into the AssemblyLine when it is launched. When an AssemblyLine executes, the built-in behavior steps through its AL Component List starting with a

Feeds Connector. This Connector returns an entry for processing which is then passed to the first component in the *Flow* section. This data is then passed from one component to the next, going down the *Flow* list until it reaches the end. At this point, control returns to the *Feeds* Connector, which can then get the next entry to be handled. Cycling continues until no more data is returned by the *Feeds* Connector.

The next section, called *Flow*, is where data transport and manipulation is carried out. Components in this part of the AssemblyLine will be writing, deleting and looking up data in connected systems, as well as performing transformations on attributes as the Entry (the Java bucket) flows down the list.

Below the Component List is the *Work Entry* list. As you select which attributes are to be read in by AssemblyLine components, they appear in this list along with the name of the component responsible for them.

To the right of these lists is where the details of the currently selected component is displayed. This area is blank right now, until you add your first Connector.

At the top of this details area are the AssemblyLine tabs. These tabs give you access to various aspects of this data flow:

Screen capture filename: GettingStarted-23.eps



Depending on your screen resolution, there may not be space on screen to display all the tabs. Whenever tabs are not visible due to screen size, buttons are provided to scroll through them.

Screen capture filename: GettingStarted-22c.eps



These tabs are:

Hooks The AssemblyLine Hooks tab enables you to write scripts to be invoked at various points during execution. We will take a closer look at Hooks later, but for now let's examine where the AssemblyLine allows you to insert your own logic:

- **Prolog - Before Initialization:** Before Connectors are initialized, allowing you to reconfigure your Connectors before they fire up their connections.
- **Prolog - After Initialization:** After Connector initialization, but before the first cycle begins. The term "cycle" here means a single pass through all the components in the AssemblyLine's AL Component List.
- **On Start of Cycle:** At the start of each AssemblyLine cycle.
- **Epilog - Before Close:** Before Connectors are closed.
- **Epilog - After Close:** After Connectors are closed.

- **On Success:** In case the AssemblyLine completed successfully.
- **On Failure:** If the AssemblyLine aborted due to an error.
- **Shutdown Request:** When the AssemblyLine is asked to terminate by some external event, so you can engineer a graceful shutdown.

Data Flow

This tab contains the AssemblyLine components list.

Config...

Gives you a number of configuration parameters for this AssemblyLine.

Operations

This is where you can define Operations for this AL, including the Input and Output Attributes for each operation

Checkpoint

Although limited in applicability, this tab lets you configure your AssemblyLine for restart in the case of abnormal termination.

Sandbox

Allows you to select the components you want to include during AssemblyLine recording or playback.

Logging

For defining logging parameters for this AssemblyLine. These are used *in addition* to those defined under the **Server->Logging** item in the Config Browser.

Description

This tab provides a text box where you can write documentation for this AssemblyLine.

The AssemblyLine **Config...** tab provides you with a number of parameters for controlling your data flow, such as error tolerance, or limiting the number of AssemblyLine iterations — very useful when developing and testing a solution that uses large data sets.

Just below the AL Component List are four buttons:

Screen capture filename: GettingStarted-24.eps



These buttons perform the following actions:

Add Component

Adds a new component to the AssemblyLine. This can also be done by dragging a pre-configured component from the Config Browser into the AL Component List.

Delete selected object

Removes the currently selected component (or components) from the AssemblyLine.

Rename selected object

Enables you to change the name of the currently selected item.

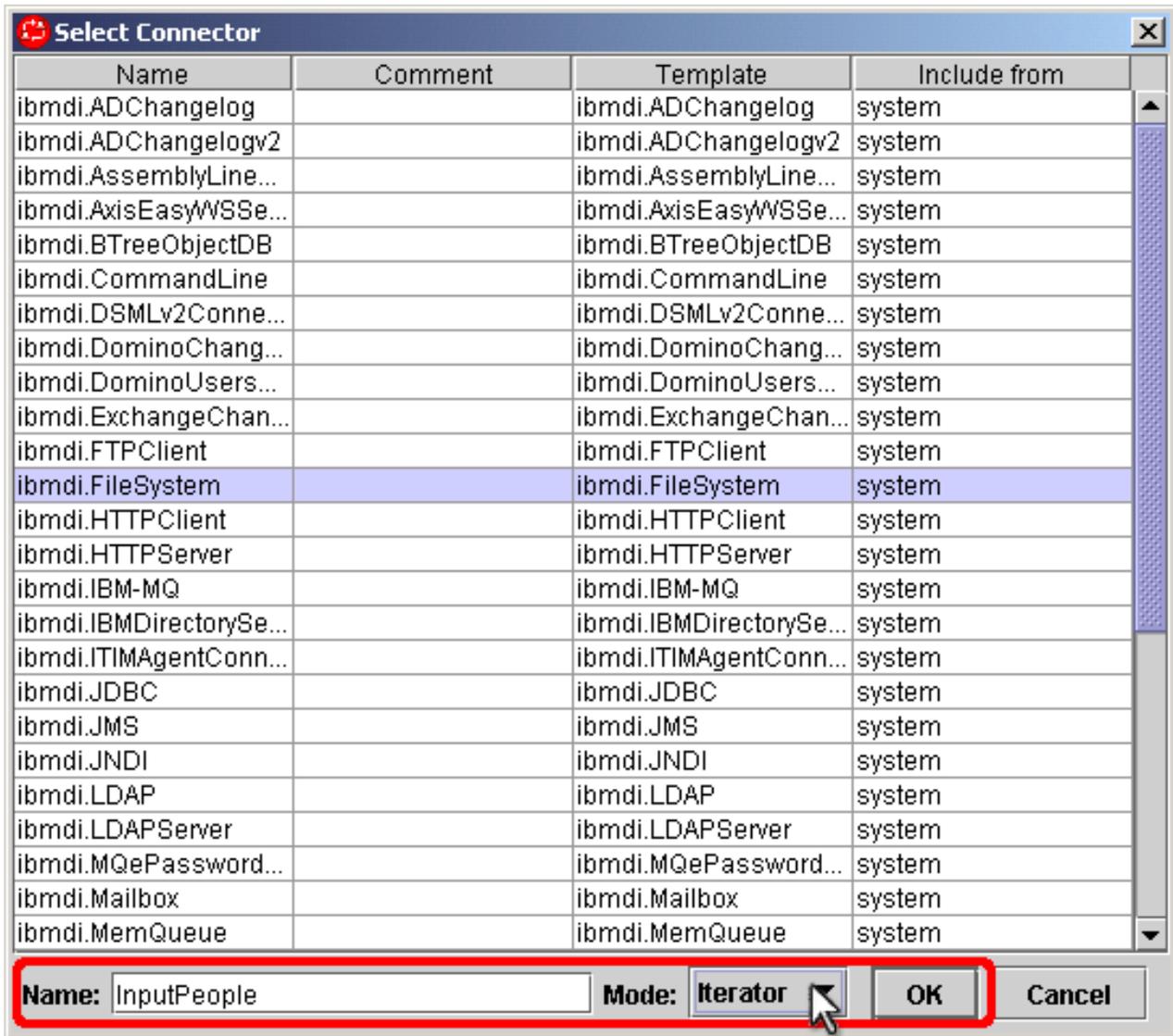
Copy to Library

Makes a copy of the selected component and drops it into the corresponding Library folder of the Config Browser.

The ordering of components in the AL Component List is significant since the built-in AssemblyLine behavior will execute them from the top-down. If you want to move a component in the list, simply select and drag it to the desired position.

Adding the Input Connector

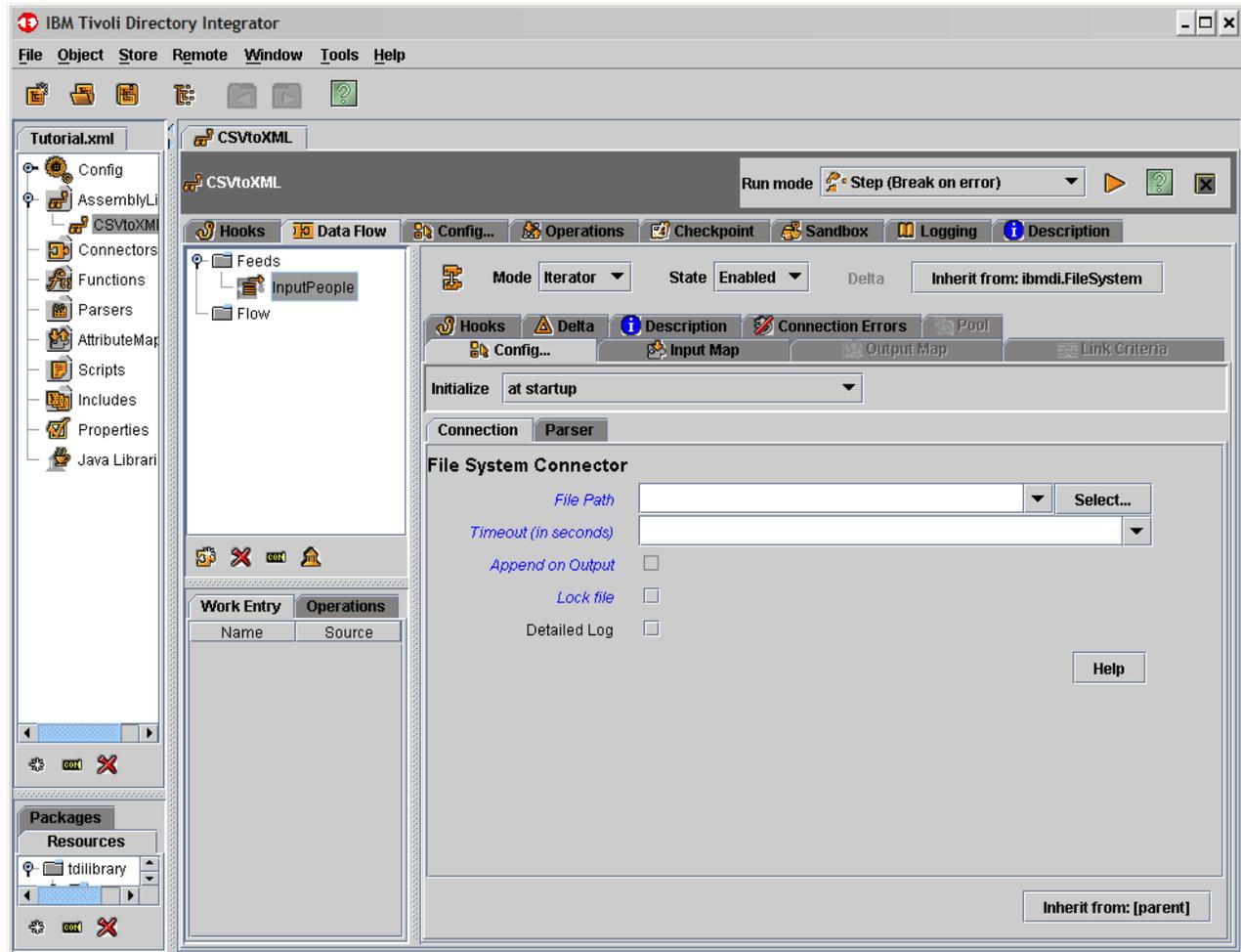
As mentioned in the previous section, you have to set up a feed for your AssemblyLine in order to get any processing done. You'll do this by adding your first Connector and then configuring it to appear in the *Feeds* section of the AssemblyLine. Click the **Add** button under the AL Component List and select **Add new Connector**. The system will present you with the New Connector dialog.



Choose the **ibmdi.FileSystem** Connector from the list and name it **InputPeople**. In order to get your Connector into the *Feeds* section of your AssemblyLine, click **Mode** and choose **Iterator**.⁹

Getting back to the exercise again, once you have selected, named and set the Mode for your Connector, click **OK** to confirm your choices. This new Connector appears in the AL Component List under the *Feeds* section. Notice how the specifics of this Connector are now shown in the Details display area to the right of the list.

9. You probably noticed that there are only two modes available for the FileSystem Connector: AddOnly and Iterator. Most Connectors only support a subset of modes, and choices presented in the Config Browser reflect this.



If you later want to change the type of Connector, simply click on the **Inherit from:** button at the top right-hand corner of the details pane and choose a different one to inherit from.

When setting up a new Connector, you generally follow three basic steps: *Configure*, *Discover* and *Map*:

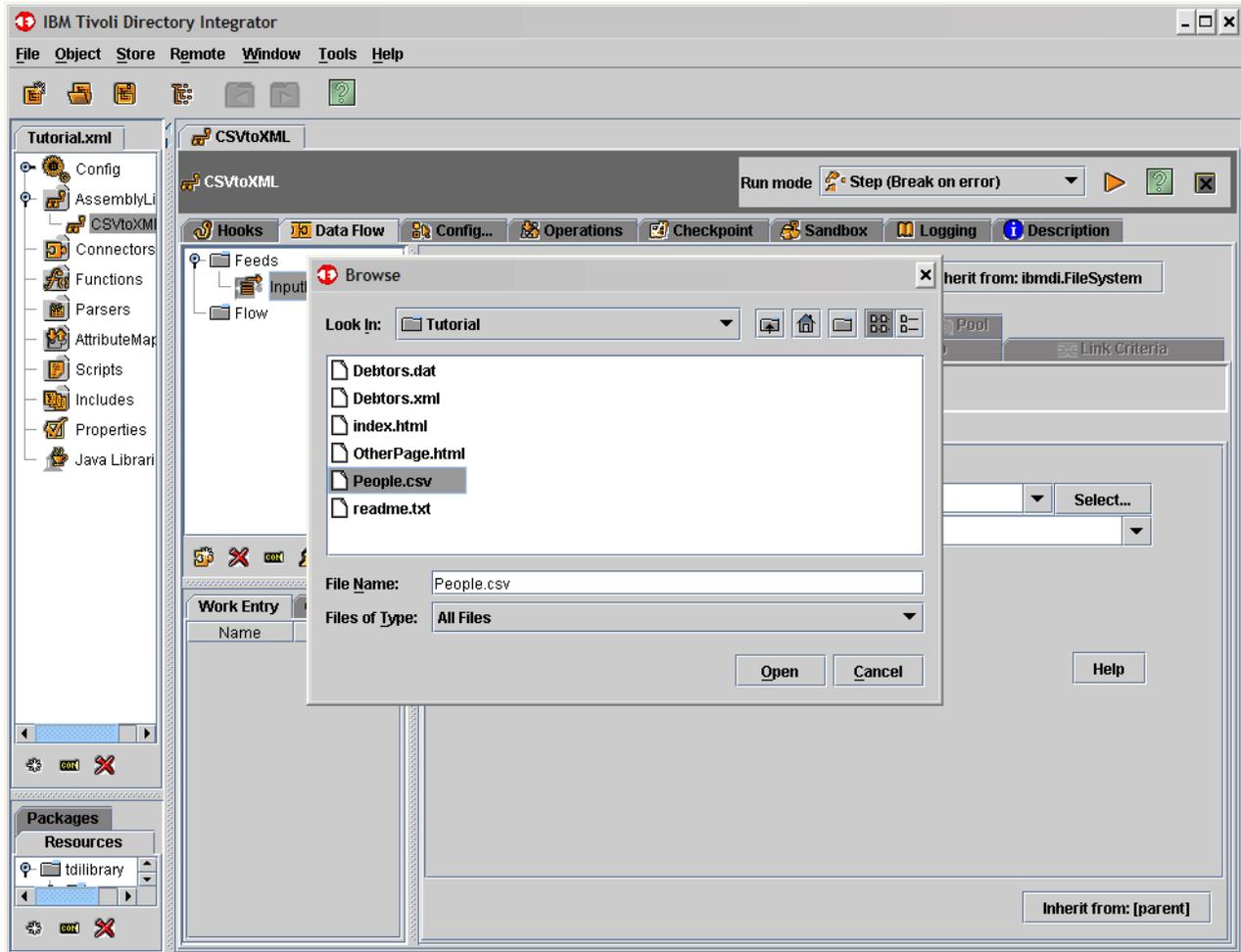
- *Configure* means setting up the parameters necessary to connect to the desired data source, possibly choosing a Parser to deal with byte stream data.
- Once the Connector is configured, the next step is to *Discover* which attributes are available in the data source. This set of available attributes is called the Connector *Schema*.
- Finally, you select which of these attributes are to be brought into the AssemblyLine for processing, also called the Connector *Input Map*¹⁰.

Returning to the example, you now need to configure your new Connector. Make sure the **Config** tab is selected in the Connector details display (as shown above). This tab is closely tied to the data source you are connecting to, and is different for each type of Connector.

10. Of course, if you are configuring an output Connector, you will define an *Output Map* instead.

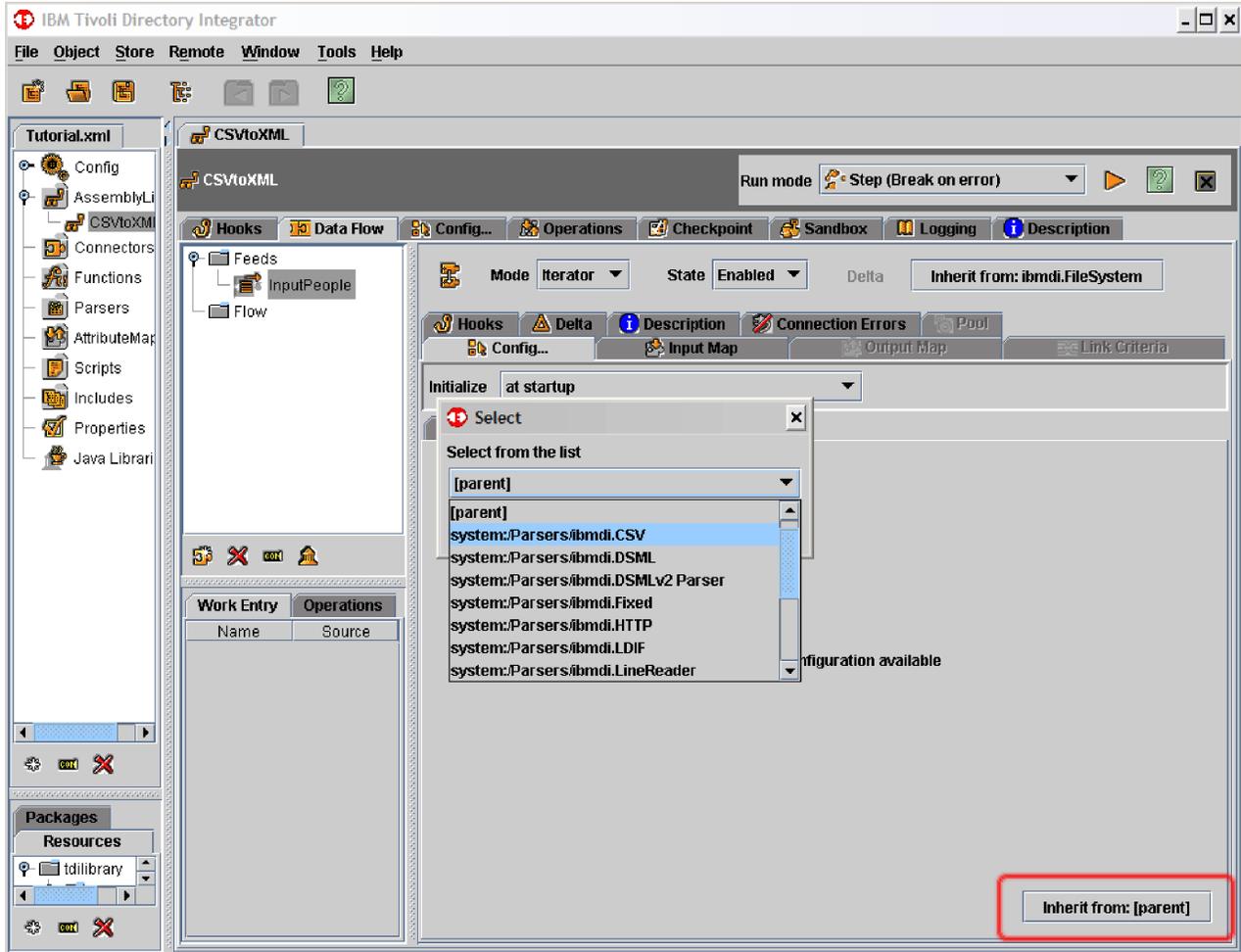
The File System Connector that you just added requires you to enter the path of the file you want to write to. Using the **Select...** button next to the **File Path** field, select the *People.csv* file in the examples/Tutorial sub-directory. Note that depending on your screen resolution, you might have to scroll the pane horizontally to access the **Select** button

Screen capture filename: GettingStarted-27.eps



Since the FileSystem Connector works with unstructured data (byte streams), you also have to assign it a Parser. This is done by first selecting the Connector's **Parser** tab and then clicking the **Inherit from:** box at the bottom of this tab.

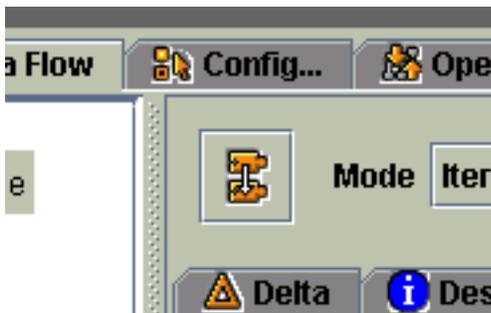
Screen capture filename: GettingStarted-27b.eps



Select **ibmdi.CSV** Parser and press **OK**.

Note: Another way to change the inheritance settings for Parser, as well as other Connector tabs, is by using the Inheritance dialog. This dialog is activated with the **Inheritance** button at the top of the Connectors details pane.

Screen capture filename: GettingStarted-28.eps



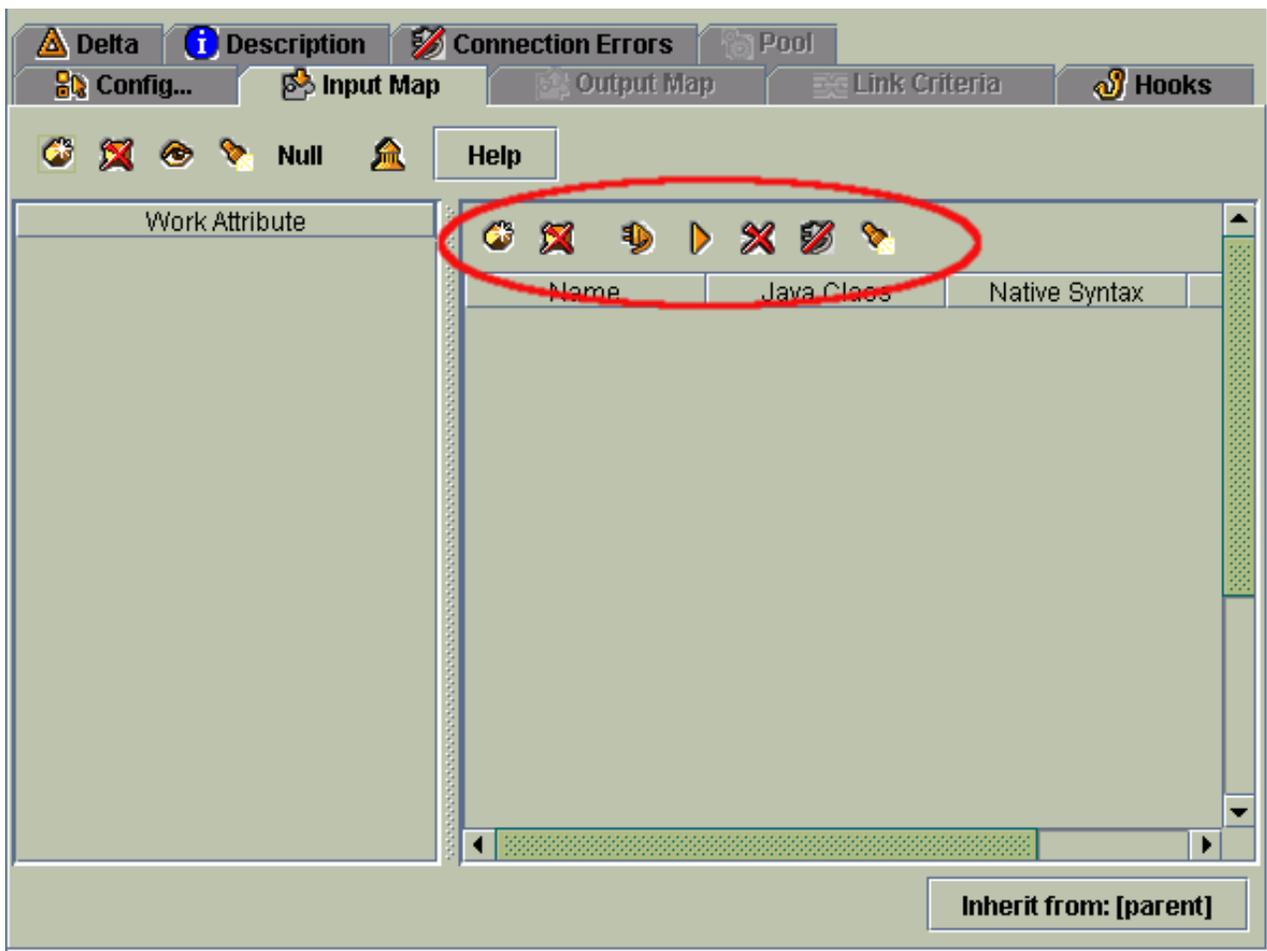
Finally, if you have a pre-configured Parser in your Component Library then you can drag it from the Config Browser and drop it on the **Inherit from:** box at the bottom of the Parser tab (not to be confused with the *parent* **Inherit from:** button at the top right-hand corner of the Connector Details

pane). All the Connector tabs represent different aspects of the Connector setup, and can be configured using the same methods described here for Parsers.

With the Parser in place, your Connector is now configured. Your next task is to test the Connection parameters and discover the list of available attributes in this data source. This is done in the **Input Map** tab¹¹.

The Input Map (and Output Map) tab contains a row of buttons which you will use to test your connection and discover the Connector Schema — the schema of the CSV input file in this example.

Screen capture filename: GettingStarted-28a.eps



These buttons work in a similar fashion regardless of the type of Connector you are using:

Add an attribute to the Schema

Sometimes you do not have access to a data source during development. Instead, you use this button to manually add attributes to the Connector Schema.

¹¹ You probably noticed that there is also a tab called **Output Map**. Of course, since this Connector is in an input Mode, only the **Input Map** is enabled.

Remove selected attributes...

Removes the attributes you have selected from the Connector Schema.

Connect to the data source

This button tests your configuration parameters by firing up the Connector and having it bind to its data source. The result of this and other data access operations is displayed in the area next to the button row.

Read the next entry

Once you have connected to your data source, each time click on this button will cause an entry to be read from the data source and then used to populate the Connector Schema.

Remove all schema attributes

Pressing this button will clear the Connector Schema, making it "forget" which attributes have been either manually added or discovered.

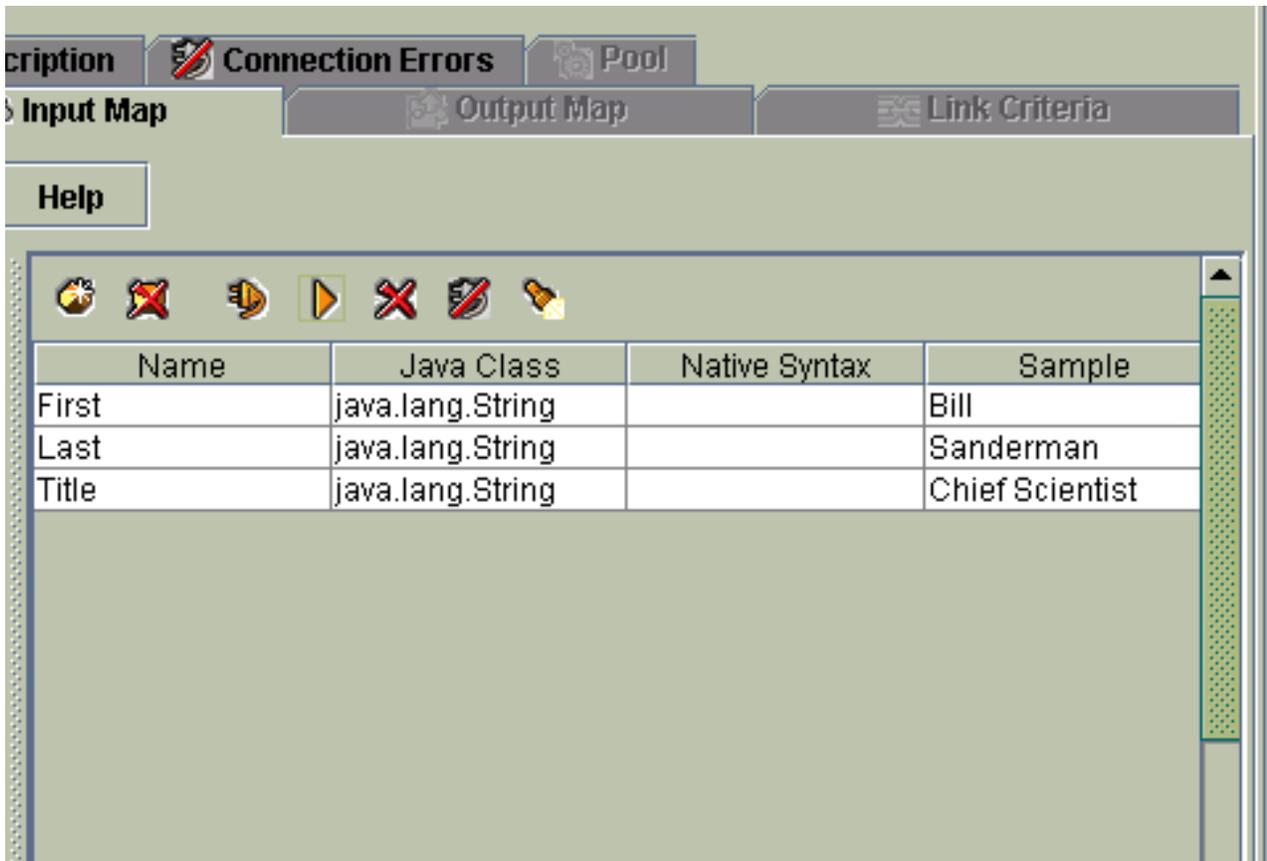
Close the connection...

Closes the connection to the data source. Note that the connection is automatically closed for you if you close the AssemblyLine Details window or run the AssemblyLine.

Discover the schema...

This button causes the Connector to request the schema from the connected data source. This will provide you with the Native Syntax for each attribute, and may be necessary to discover the complete attribute list (for example, when working with a directory where attributes may be optional and not appear in entries examined using the **Read next** button). Note that not all data sources provide a schema discovery mechanism, and you may be limited to stepping through the data instead.

Test your Connector now by pressing the **Connect** button. If your Connector manages to find and open the input file then you will see the message Connection established in the area to the right of the button row. Now click the **Read the next entry** button, instructing your Connector to read an entry from the data source and add this information to the Connector Schema list.



The Connector Schema you have just discovered is displayed in tabular form with the following columns:

Name This is the name of the attribute as it appears in the data source.

Java Class

Each Connector converts attribute values between data source native types and Java objects, which is the internal representation used by IBM Tivoli Directory Integrator. Here you will see the Java objects used for each attribute.

Native Syntax

If you press the Discover Schema button, then this column will contain the native type used by the data source to store each attribute.

Sample

Displays actual data values being read. This helpful feature allows you to control that you are getting (and possibly parsing) the data as expected.

Each row in the Connector Schema list represents a single attribute that is available in the connected data source.

You have now completed two of the three steps in setting up the Connector: You have *configured* it (and tested it) and have *discovered* the schema available in the connected system. Your next task will be to *map* the attributes that you want brought into the AssemblyLine for processing.

Mapping Attributes Into The AssemblyLine

Attribute Mapping is the operation of selecting and possibly transforming the data that will be moving between a data source and the AssemblyLine. You saw in the previous step how IBM Tivoli Directory Integrator not only discovers the schema of a connected system for you, it also automatically converts the data to Java objects. So why is Attribute Mapping necessary?

Although IBM Tivoli Directory Integrator can discover what is available in the data source, the system has no preconceptions of which attributes you need or what format you need them in. So at the very least, you must select those attributes you want to use. Furthermore, some of the attributes might have to be computed, combined or converted to a different format or type.

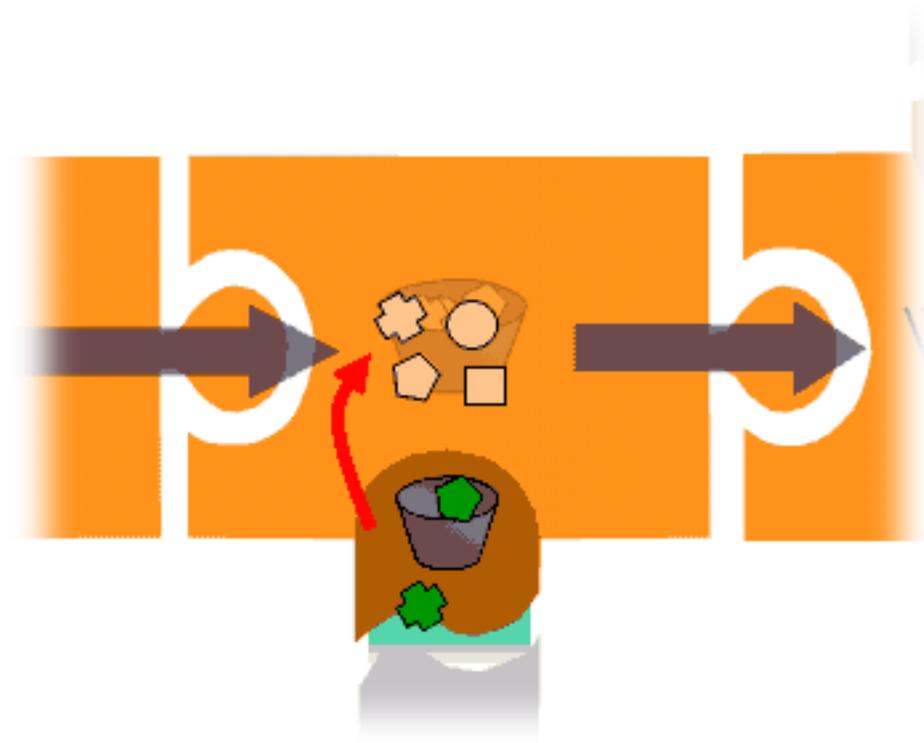
To get an idea of how Attribute Mapping works in IBM Tivoli Directory Integrator, take another look at the AssemblyLine image on page 8. As discussed briefly before, all AssemblyLine components reflect the kernel/component architecture of the system. As a result, they are made up of two parts:

The component Interface (e.g. Connector Interface, Parser Interface, and so forth) For Connectors, this is the data source "intelligence", built to access data via a particular protocol or API. The Interface part of any component is the part that is platform, technology, format or data source specific.

The AssemblyLine component (called *AL Connector* for Connectors, *AL Parser*, and so forth)

Connectors, like all components, get wrapped when dropped into an AssemblyLine. This AssemblyLine wrapper provides generic (kernel-based) functionality like Attribute Maps, as well allowing the built-in AssemblyLine behavior to link components together and drive them for you. This is also the part of the component that holds any custom logic you have added, like attribute transformations, data filtering, error handling or flow control instructions.

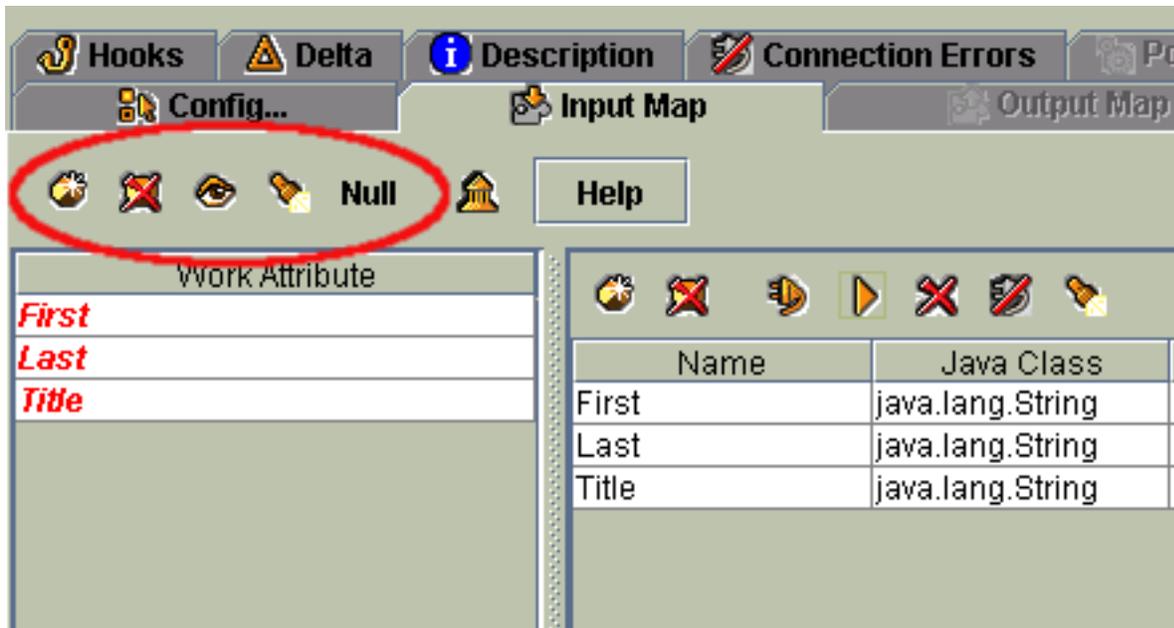
By setting the Mode of your AL Connector, you are telling the AssemblyLine what operation you want carried out by its Connector Interface. Since all components of the same type share a common set of features, this means that you can swap out the Connector Interface of an AL Connector without touching any of your customization.



As also discussed previously, all data is handled in a storage object called an Entry. Information being passed down an AssemblyLine is kept in a special Entry object called **work**. Information passed to or from a data source is kept in a local Entry object called **conn** that is managed by the Connector Interface. Attribute Mapping is the process of copying Attributes from one Entry object to another, and is shown by the curved arrow in the above diagram. If your Connector is in an input Mode, then you must define an Input Map that details which attributes you want merged from the Connector's **conn** object into **work**. An Output Map defines which of the **work** attributes that you want sent to the local **conn** object, which in turn is used by the Connector to perform the output data operation.

Since your Connector (**InputPeople**) is in an input mode then you must set up an Input Map. This is done by selecting one more Attributes in the Connector Schema that you just discovered and then dragging them to the Work Attribute list¹².

12. Selection lists in IBM Tivoli Directory Integrator, like the AL Component list, the Connector Schema and Attribute Maps, support multiline selections. You can do this by using the Ctrl button while clicking on individual line items, or the Shift button for selecting a from-to range. The Ctrl + A is a keyboard shortcut for selecting all elements in a list.



In addition to dragging and dropping attributes, you also have a number of buttons at the top of the Attribute Map for creating and manipulating your Map:

Add Attribute

This button allows you to manually add attributes to the Map.

Remove Attribute

Removes the selected Attribute (or Attributes) from the Map.

Toggle View

Toggles the Attribute Map view between List, Detail and Schema displays.

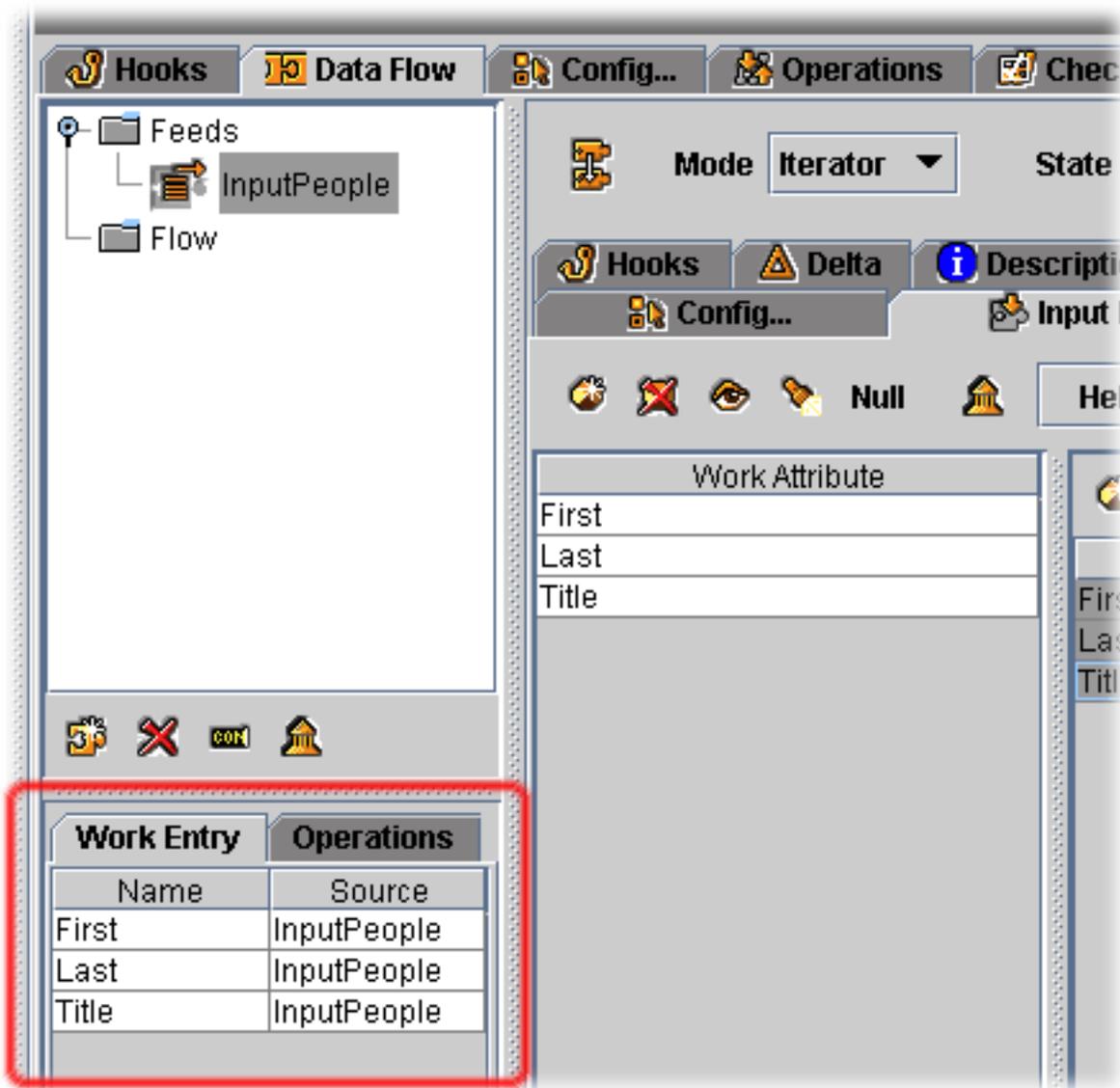
Quick Discovery

This is a convenience button, and is equivalent to using the **Connect** and **Read next entry** buttons in the Connector Schema button bar.

Null Behavior

This button allows you to define how the Connector will deal with missing Attributes; for example, by deleting them from the mapping operation, or giving them a user-defined default value.

Whenever you add Attributes to the Input Map, they also show up both in the AssemblyLine's **Work Entry** display, located just below the AssemblyLine Connector list.

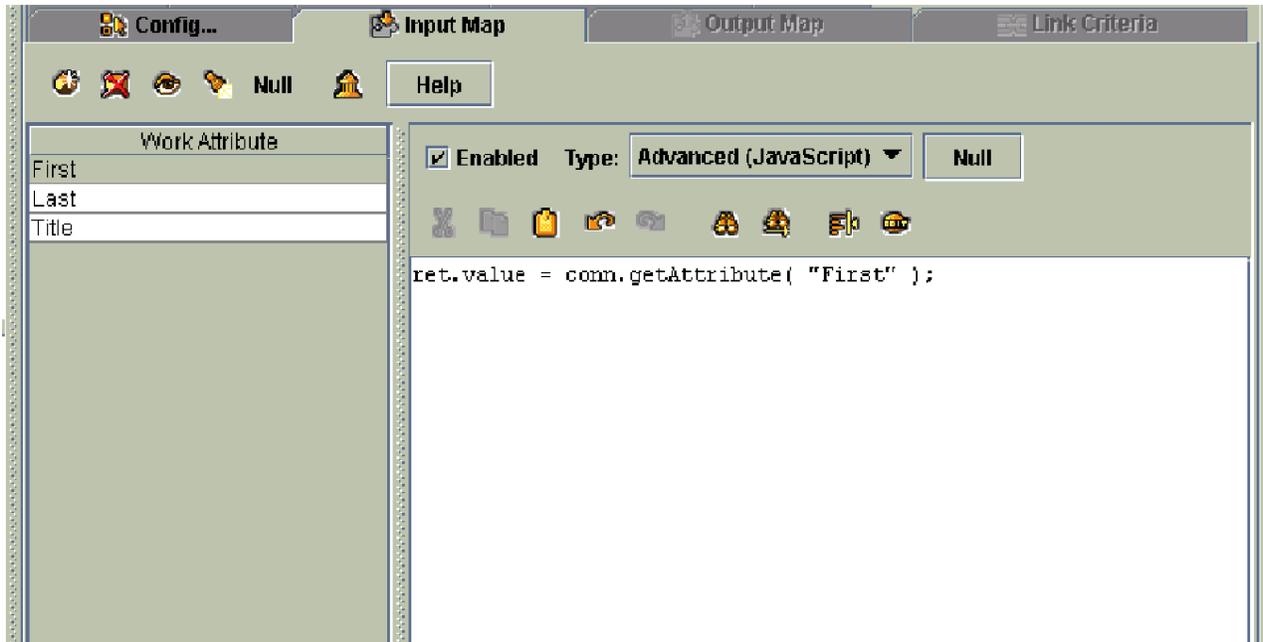


The Work Entry display is like a window into your data flow, showing you which attributes are being mapped in from connected systems (or created by you), as well as the name of the Connectors responsible for them. Any Attributes defined in an Input Map in your AssemblyLine appear in the Work Entry window.

Note: There is also an **Operations** tab that displays Input Mapped parameters for an AssemblyLine with Operations defined. However, this is an advanced topic that is not covered in this text.

Use the **Toggle View** button to view the **Schema** display. Once you've completed your first Input Map, your screen should look something like this:

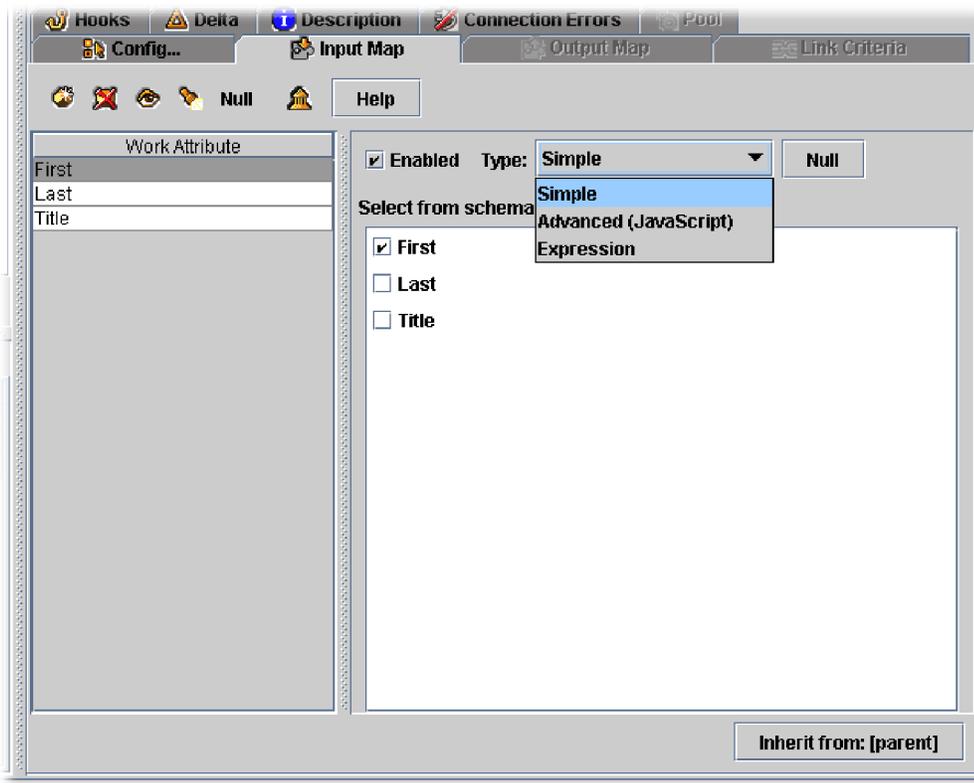
Screen capture filename: GettingStarted-36.eps



This Attribute Map is what is called a *simple* Map, meaning a one-to-one direct transfer of attributes and values from the Connector cache (the **conn** object) to the Assembly Line's **work** Entry. Sometimes you need more control over how data is mapped. To change the mapping of an attribute, you must first select it in the Map.

Selecting an Attribute in the Attribute Map brings up its mapping settings¹³.

13. Attribute Map settings fill the same screen area previously occupied by the Connector Schema. In order to restore the Schema view, use the Toggle View button above the Attribute Map.



The settings display tells the user how this Attribute is being handled, and includes the following controls:

Enabled

You can exclude an Attribute from the Map by clearing this check box.

Type This menu allows you to specify how this Attribute map is to be carried out. The options are:

- Simple - Simple type means that the mapping is done by copying values from the selected Attributes in the list
- Advanced (JavaScript) - Advanced mapping, on the other hand, is done by executing JavaScript, where values can be computed.
- Expression - Allows you to define a TDI expression for evaluation. For more information about TDI expressions, see the *IBM Tivoli Directory Integrator 6.1: Users Guide*.

Null behavior

This button allows you to override Null Behavior for this attribute.

Select from schema (list)

Simple mapping is defined by the selections made in this list. Most Attributes are mapped to a single source; however, you can select multiple source Attributes in this list, resulting in a multi-value mapping.

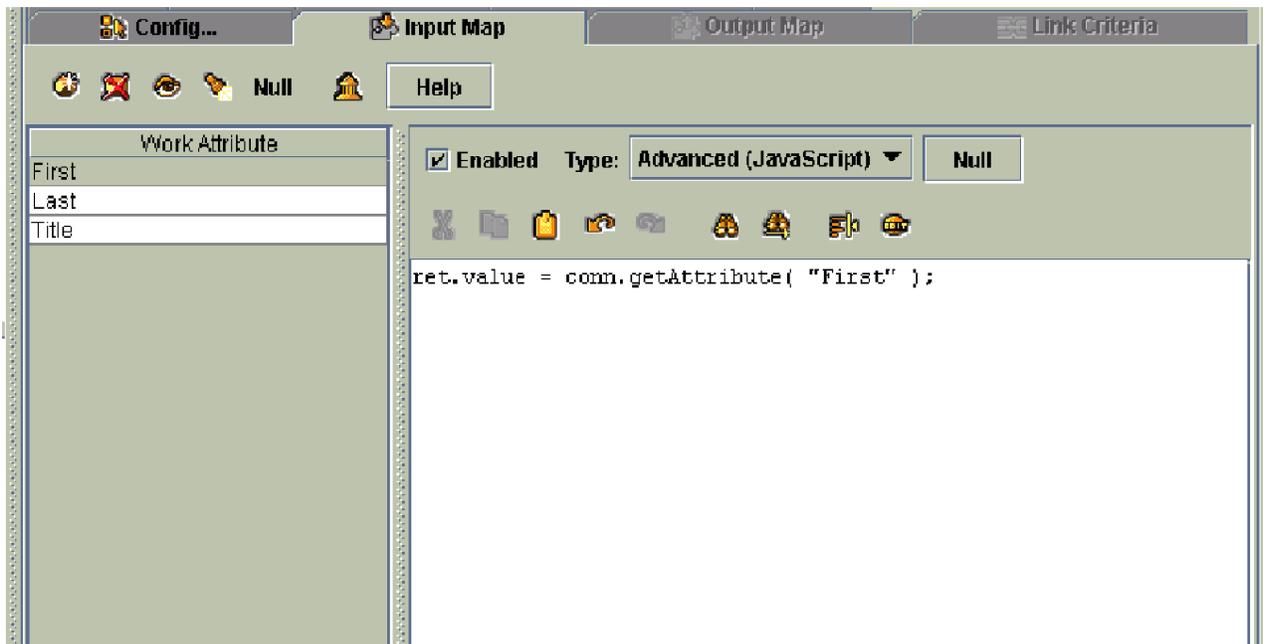
According to best practices, solution customization is divided into two areas: attribute manipulation and flow control. Attribute manipulation is done, as much as possible, in Attribute Maps. This makes your solution easier to read and maintain, ensuring that any attributes handled in the flow show up in the Work Entry display.

To get your feet wet with attribute manipulation, select the "First" Attribute (if it is not already selected) and then select **Advanced (JavaScript)** from the **Type** menu. The lower part of the Attribute Map settings pane changes to a Script Editor window where you can enter the following snippet of JavaScript:

```
ret.value = conn.getAttribute("First");
```

The purpose of an Advanced Map is to return a value for the selected Attribute, accomplished by setting the `ret.value` variable at some point in the script. As mentioned earlier, there is a local storage object in each Connector called **conn** which is used to cache read operations. This object is available as a script variable for use in Advanced Maps and gives you direct access the Connector's local cache. The above code sets `ret.value` to the Attribute called **First** that has just been read into **conn**, instructing the system to copy its values during the mapping operation — in effect, duplicating the behavior of simple attribute mapping.

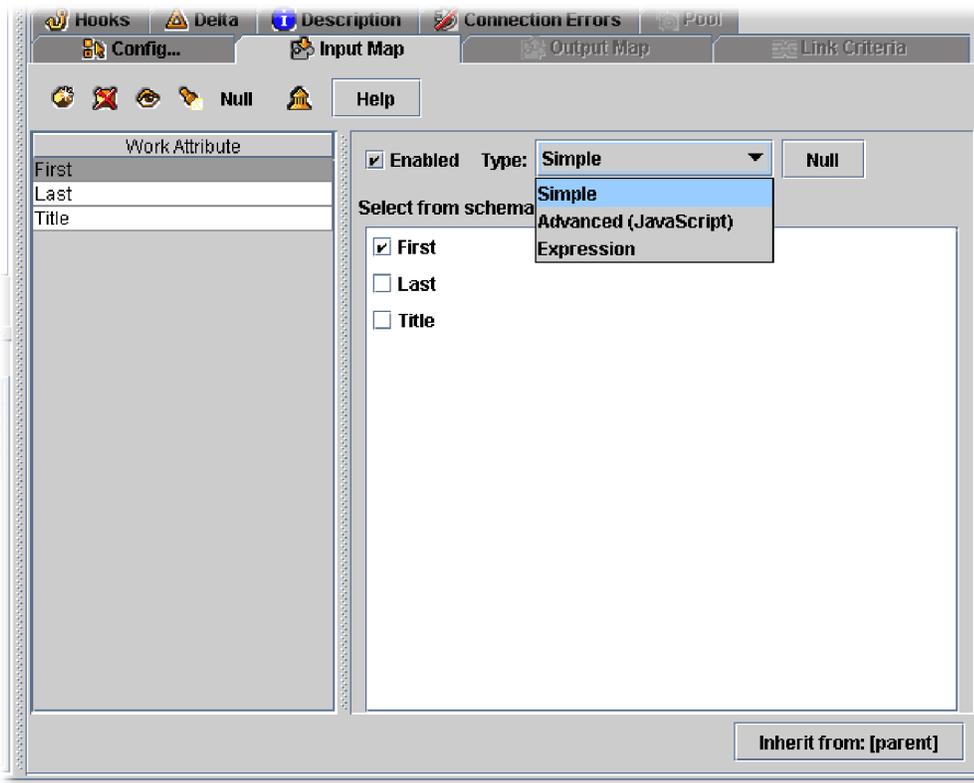
Screen capture filename: GettingStarted-36.eps



Restore the simple mapping of this attribute by selecting **Simple** from **Type**. Not only is simple attribute mapping easier to read, but it's faster than invoking script code.

Let's practice customization on something more productive than simulating a simple map. If you remember the description of our example integration problem on page 6, there are a few attributes that do not exist in the input file. These will have to be created by your solution; specifically, in one of your Attribute Maps.

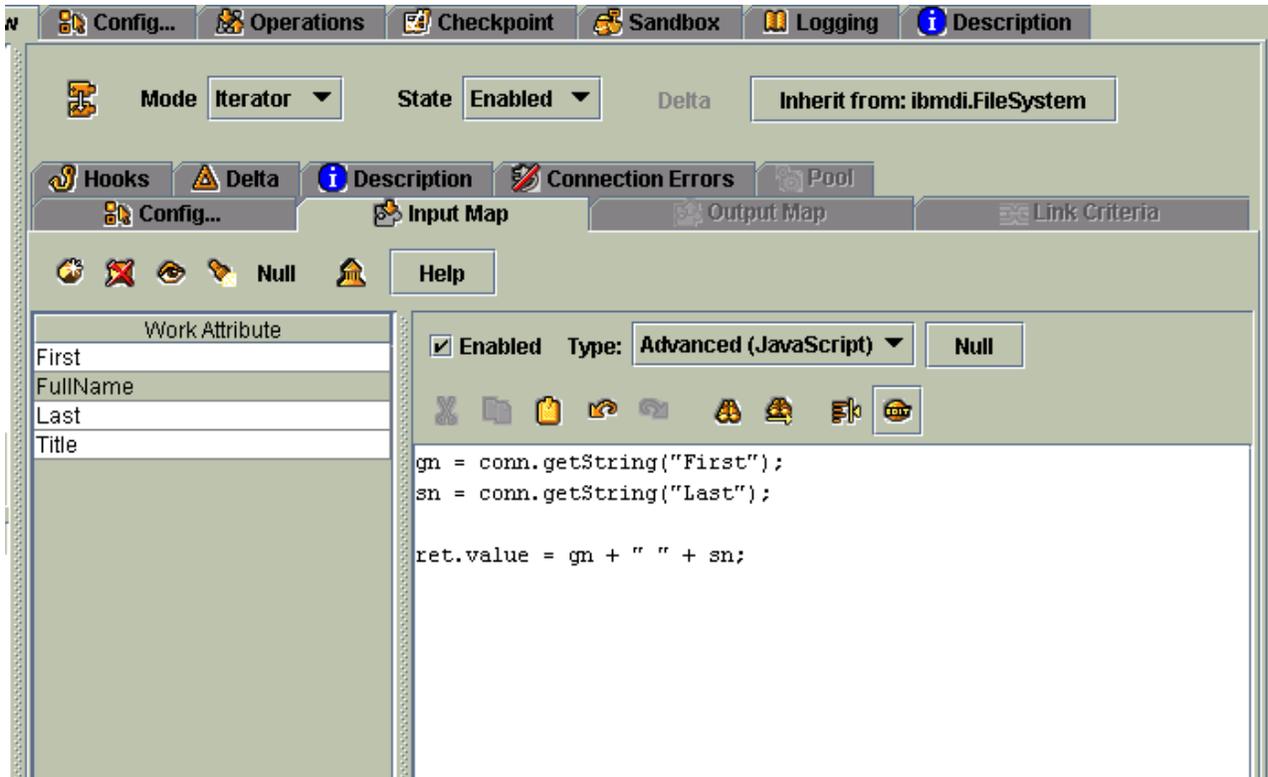
Start by adding a new Attribute to the Input Map. Do this by pressing the **Add a new Attribute...** button in the toolbar at the top of the Input Map.



In the dialog box that appears, enter *"FullName"* and click **OK** (or press the Enter key twice: first to acknowledge your field editing and then to accept and close the dialog box).

Your new Attribute appears in both the Input Map *and* the Connector Schema list. This is because IBM Tivoli Directory Integrator assumes that you want to use simple mapping (the most effective method) to retrieve the Attribute's value. Of course this won't work since there is no **FullName** Attribute in the CSV input file.

Instead, you must select this new Attribute from the **Type** menu and select **Advanced (JavaScript)**. This opens the Script Editor window again where you can enter several lines of script such as those shown below.



Here is the code so you can cut-and-paste it into the editor if you want:

```
gn = conn.getString("First");
sn = conn.getString("Last");

ret.value = gn + " " + sn;
```

Note: Pasting some characters may not work because they are represented differently in the documentation file (PDF) than their simple character counterpart as used in the script. Not all script lines shown in this guide can be simply copied and pasted into TDI. If you receive error messages of the this type:

```
com.ibm.jscript.parser.ParseException: Syntax error at line 1, column 38. Invalid '\u2019'
```

this indicates that some character (a single quote in this above example) is invalid. Just retype these characters and try again.

The first two lines above retrieve the values of the Attributes named **First** and **Last** as Java Strings (`java.lang.String`), and stores them in two new variables called `gn` and `sn` respectively: The final statement returns the value of these two local variables concatenated together with a single space between them.

Notes:

1. This Attribute can also be computed using the following TDI Expression:
{conn.First} {conn.Last}

It is unnecessary to set "ret.value" to map the Attribute. The result of the Expression is used as the Attribute's mapped value.

2. You can also create this attribute in the output Connector you are about to add. However, because you need it later in your AssemblyLine, it is necessary to have this code in the Input Map of the **InputPeople** Connector.

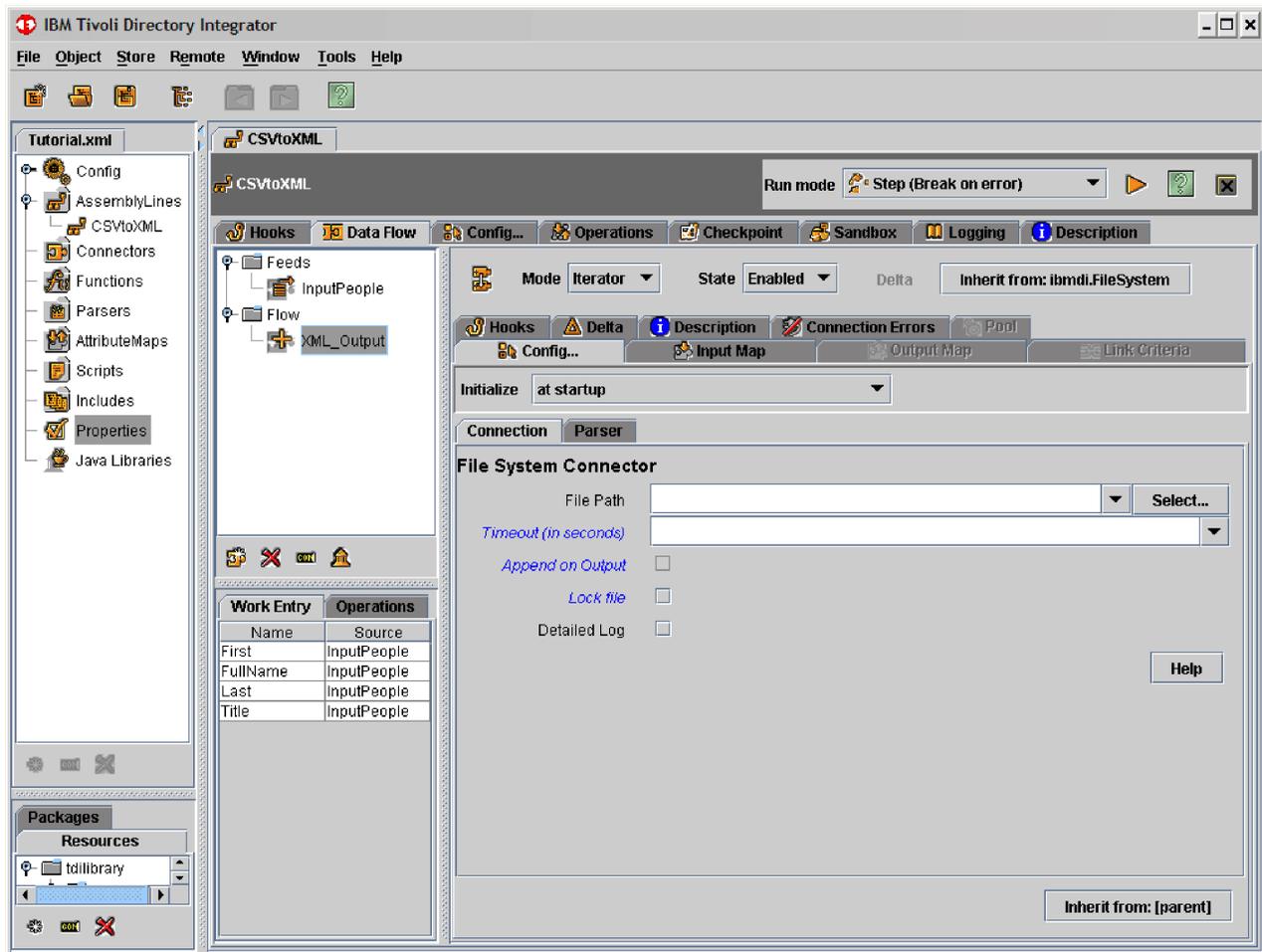
The Entry *Feed* to your AssemblyLine is now finished. You have configured a Connector; tested it by connecting to the input source and discovered the schema of the CSV file; Then you selected the attributes you want to process in your data flow; and even computed a new using custom script. Time to complete the first phase of this exercise.

Adding the Output Connector

You will complete your AssemblyLine by adding a *Flow* component to process the Entries coming out of your Iterator *Feed* (i.e. the CSV FileSystem Connector in Iterator Mode). The target system for the AssemblyLine is to be an XML document, so you need to add another FileSystem Connector by again clicking the **Add** button in the button toolbar at the bottom of the AL Components list. Name this Connector **XMLOutput**. Choose the **FileSystem** type again and set the Connector mode to **AddOnly**.

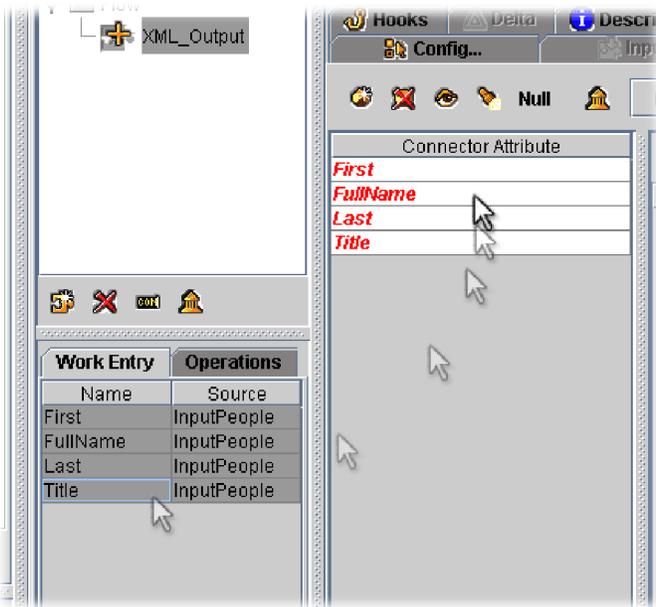
With this Connector still selected, configure the Connection parameters by setting the output file name to "*Output.xml*" and writing it to the same directory where the input file is located.

Now click on the Parser tab and select the XML Parser.



Just as you specified which attributes your *Iterator* (**InputPeople**) was to bring into the *AssemblyLine*, you must now also tell your new output Connector what you want written to the XML document. Do this by clicking the **Output Map** of your **XMLOutput** Connector. Now select all the Attributes in the Work Entry display (e.g. select one and press Ctrl + A) and then drag them to the Output Map.

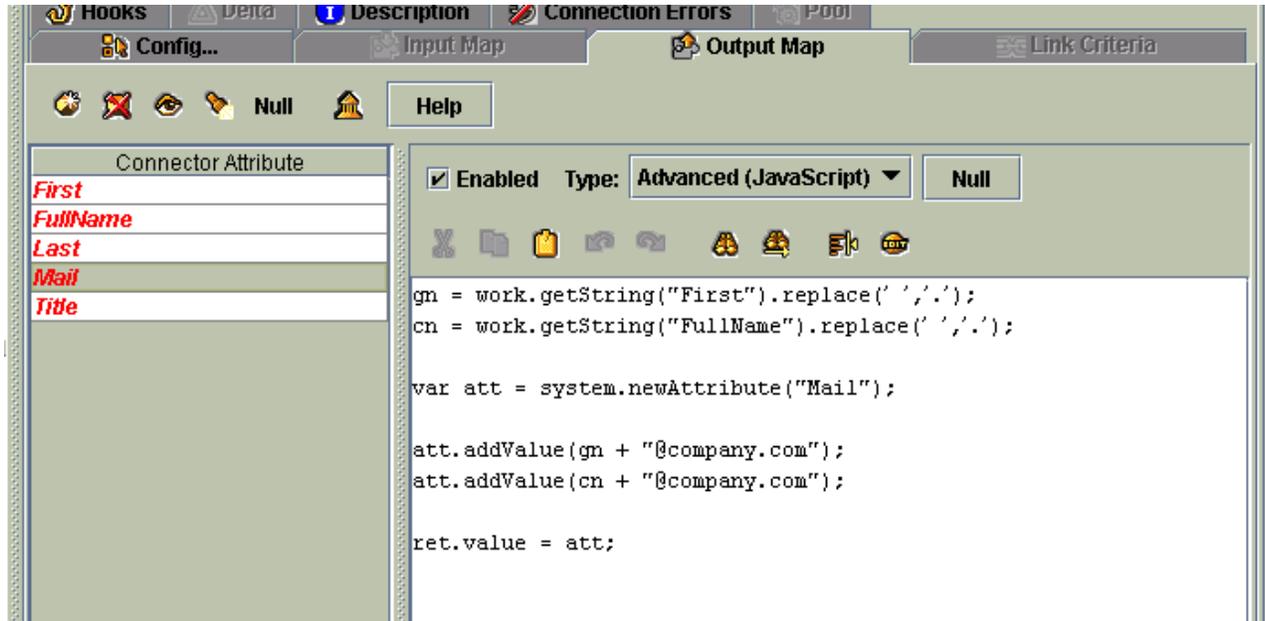
Screen capture filename: GettingStarted-64.eps



Finally, you need to create another computed Attribute, **Mail**, as specified back on page 6. You will base the new value of the **Mail** Attribute on the one being computed previously during CSV input: **FullName**. Furthermore, since people often have more than one e-mail address, you will compute two values, making **mail** a multi-value attribute.

To do this, click the **Add Attribute** button in the toolbar at the top of the Output Map. This opens a dialog asking you to name the new attribute. Call it **Mail** and click **OK**.

Select this new Attribute and then select **Advanced (JavaScript)** from the **Type** menu. Here is the script code to use:



Let's go through this script to see what it does.

```
gn = work.getString("First").replace(' ','.');
cn = work.getString("FullName").replace(' ','');
```

These first two lines store the values of the **First** and **FullName** attributes in local variables. Notice how you use the **work** object to access data inside the **AssemblyLine**. The `.replace()` function is a standard Java String method that will swap out any spaces found with dots. You can use this function since `work.getString()` returns the value of a named Attribute as a Java String, and you need to do this since e-mail addresses cannot have spaces in them: "Peter Belamy" must be changed to "Peter.Belamy"

```
var att = system.newAttribute("Mail");
```

This next line uses a system call to create a new Attribute. The **system** object provides methods for creating many important objects, like Attributes and Entries. It also gives you access to flow control functions, as you will see later.

```
att.addValue(gn + "@company.com");
att.addValue(cn + "@company.com");
```

Now the new e-mail addresses are computed and added as values to the Attribute.

```
ret.value = att;
```

Finally, the newly created attribute is returned and used for the Output Map¹⁴.

14. The Advanced Mapping examples so far have returned an Attribute object, not just a value to use. Sometimes it's enough to simply specify the value — which can be any type of Java object. For example, if you were computing an Attribute called "EmployeeType", you could use the following script snippet to handle mapping:

```
ret.value = "Full-Time";
```

Your first data flow implementation is ready to test.

Running your AssemblyLine

In order to test your AssemblyLine, you must start up the run-time Server, point it at your Config and instruct it to launch your solution. Fortunately, the Config Editor provides an easy way to do this. Simply click the **Run** button in the AssemblyLine's button toolbar at the top of the AssemblyLine Details window (or use the Alt + R keyboard shortcut).

In this case, the returned String value is used for the mapped Attribute's value. You can also return multiple values by using a JavaScript array:

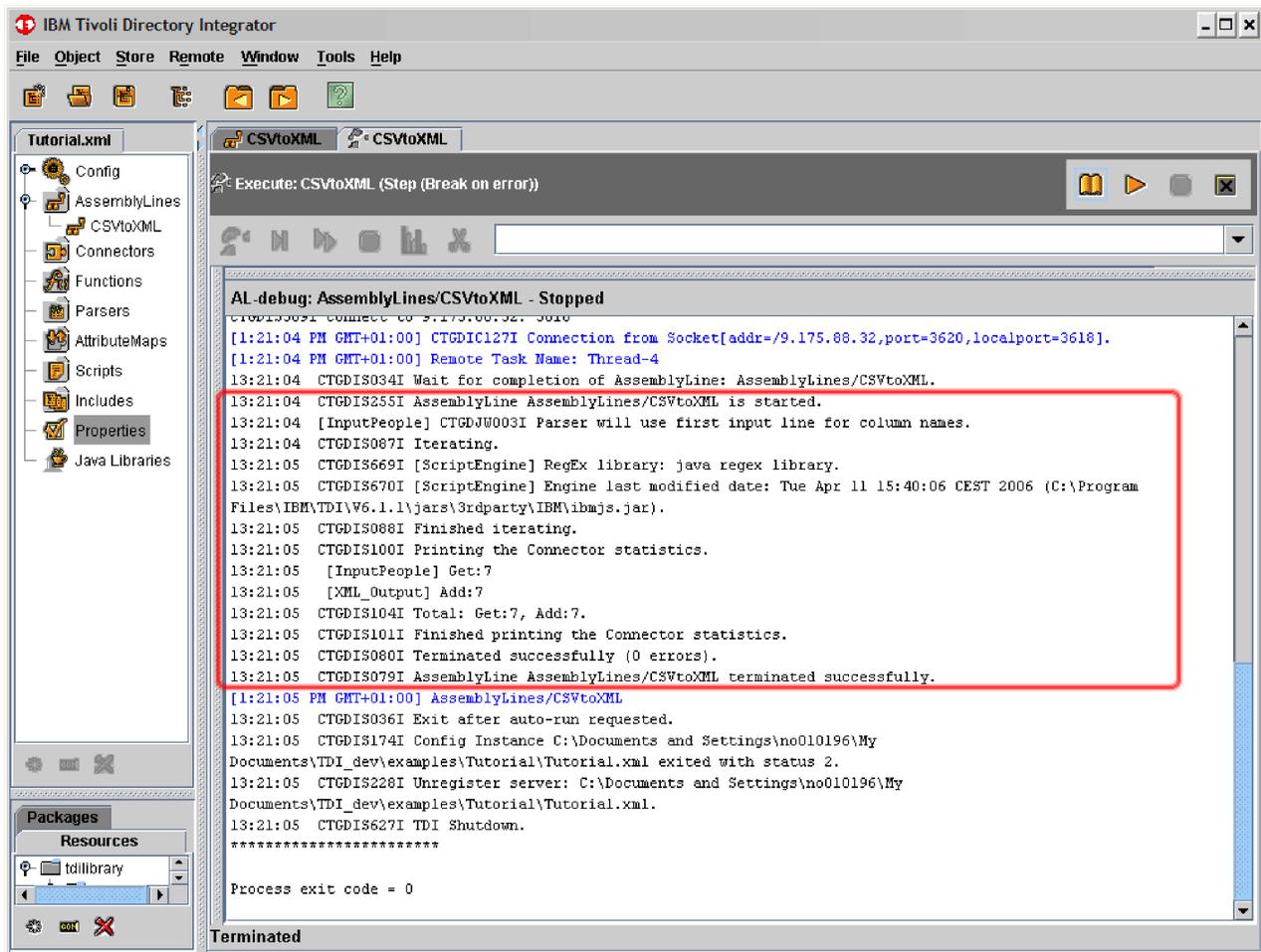
```
ret.value = ["This is the first value", "and here is the second"];
```

The above code would result in a multi-valued Attribute mapping containing two String values.

```
</Entry>
- <Entry>
  <Last>Kamerun</Last>
  <First>Mick</First>
  <Title>CEO</Title>
  - <Mail>
    <ValueTag>Mick@company.com</ValueTag>
    <ValueTag>Mick.Kamerun@company.com</ValueTag>
  </Mail>
  <FullName>Mick Kamerun</FullName>
</Entry>
- <Entry>
  <Last>Vox</Last>
  <First>Jill</First>
  <Title>CTO</Title>
  - <Mail>
    <ValueTag>Jill@company.com</ValueTag>
    <ValueTag>Jill.Vox@company.com</ValueTag>
  </Mail>
  <FullName>Jill Vox</FullName>
</Entry>
- <Entry>
  <First>Roger</First>
  - <Mail>
    <ValueTag>Roger@company.com</ValueTag>
    <ValueTag>Roger.null@company.com</ValueTag>
  </Mail>
  <FullName>Roger null</FullName>
</Entry>
- <Entry>
  <Last>Highpeak</Last>
  <First>Gregory</First>
  <Title>VP Product Development</Title>
  - <Mail>
    <ValueTag>Gregory@company.com</ValueTag>
```

When you start an AssemblyLine from the Config Editor, the system starts a separate instance of the Server, pipes the current configuration to it and instructs it to run this AL. As a result, your test behaves exactly as it will in deployment.

IBM Tivoli Directory Integrator now creates a new details tab labeled **Execute: CSVtoXML** displaying the log output of your AssemblyLine.



Aside from the **Process exit code** at the bottom that tells you that AssemblyLine stopped after reaching the end of its input data set, this log is divided into three main parts:

- Information about the version of the server that you are running.
- Description of the environment that IBM Tivoli Directory Integrator is running in, including which VM it is configured to use, and the working directory.
- Information about how your solution performed, including:
 - Specifics about tracing and API services started.
 - The Config that is being used — which in this case is shown as `<stdin>` meaning that it was piped to the runtime server from the Config Editor.
 - Messages generated during the running of the AssemblyLine and its Connectors.
 - Information about how your solution performed.

Note: Information from the Stepper/Debugger is displayed in blue. You will see these whenever you run an AssemblyLine in Step mode.

At the bottom of this last section is the message that the AssemblyLine (CSVtoXML) ran without errors. This means that you can open the output file that you specified in your **XMLOuput** Connector. Opening this file (for example, in a

browser) enables you to confirm that the AssemblyLine has actually converted the CSV input data to the XML document.

Screen capture filename: GettingStarted-44.eps

```
</Entry>
- <Entry>
  <Last>Kamerun</Last>
  <First>Mick</First>
  <Title>CEO</Title>
  - <Mail>
    <ValueTag>Mick@company.com</ValueTag>
    <ValueTag>Mick.Kamerun@company.com</ValueTag>
  </Mail>
  <FullName>Mick Kamerun</FullName>
</Entry>
- <Entry>
  <Last>Vox</Last>
  <First>Jill</First>
  <Title>CTO</Title>
  - <Mail>
    <ValueTag>Jill@company.com</ValueTag>
    <ValueTag>Jill.Vox@company.com</ValueTag>
  </Mail>
  <FullName>Jill Vox</FullName>
</Entry>
- <Entry>
  <First>Roger</First>
  - <Mail>
    <ValueTag>Roger@company.com</ValueTag>
    <ValueTag>Roger.null@company.com</ValueTag>
  </Mail>
  <FullName>Roger null</FullName>
</Entry>
- <Entry>
  <Last>Highpeak</Last>
  <First>Gregory</First>
  <Title>VP Product Development</Title>
  - <Mail>
    <ValueTag>Gregory@company.com</ValueTag>
```

Even the **Mail** and **FullName** attributes are there, computed by your script snippets.

However, one of the entries (**Roger**, outlined in the above screenshot) is incomplete. This entry is lacking both the **Last** and **Title** attributes¹⁵. If you check the input data file (see page 10) then you can see that these fields are actually missing from the input CSV file.

15. You probably also notice the text "null" appearing in the values of your computed Attributes. This is because of default Null Behavior which causes missing Attributes to be removed from an Attribute Map. These Attributes, **Last** or **Title**, were never available for Roger's Entry. When you later used the `.getString("Last")` method to retrieve the string value of **Last** in order

The easiest solution is to edit the CSV file and add the missing fields. However, few data sources give you this much control. Instead, you are going to filter the input by scripting a **Hook**.

Note: But before you do anything else, save your work in one of the following ways::

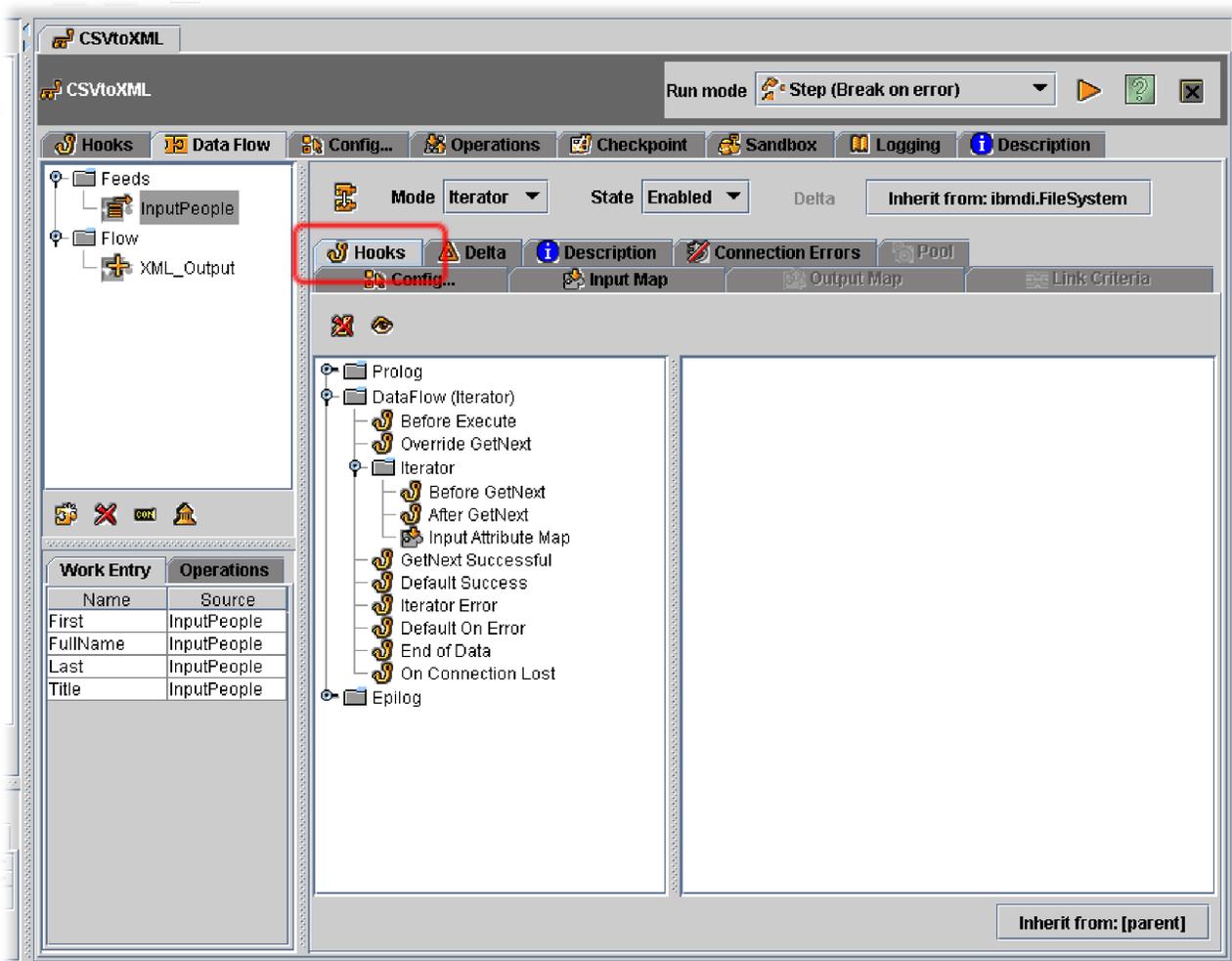
- Click **Save** button in the main tool bar.
- Select **File->Save** from the Main Menu.
- Use the Ctrl + S keyboard shortcut.

Working with Hooks

Hooks are waypoints in the built in behavior of the AssemblyLine, as well as that of Connectors and Function components where you can add your own scripted logic. In most cases, Hooks are provided for you to extend the automated workflows. However, some Hooks even allow you to override standard behavior, like the Override GetNext Hook of your InputPeople Iterator Connector. If this Override Hook is enabled, then it replaces the subsequent Connector flow up to (but not including) the Success/Error Hooks.

Getting back to the exercise at hand, your next task is to check for missing data in the CSV input file. Since this data source is handled by the Connector called **InputPeople**, select it in the AL Component List and then choose the Connector's **Hooks** tab. This will give you access to the tree-view of Hooks.

to compute the value of **FullName**, which was later used to compute the **Mail** address, the `.getString()` function could not find these Attributes. Instead of crashing your script, this function simply returns the String value "null" to indicate failure. Note that if you use the `.getAttribute()` method instead of `.getString()`, you actually get a *null* return.



Hooks are listed, from the top down, in the same order that they are executed. There are three sets of Hooks for every Connector, represented by folders in the Hooks tree-list:

In Prolog

There are at least two Hooks in this folder:

- One run just before the Connector is initialized.
- One run just after initialization.

In Iterator mode you also get two additional Hooks: Before and After the selection operation, where the Connector requests its input data.

DataFlow

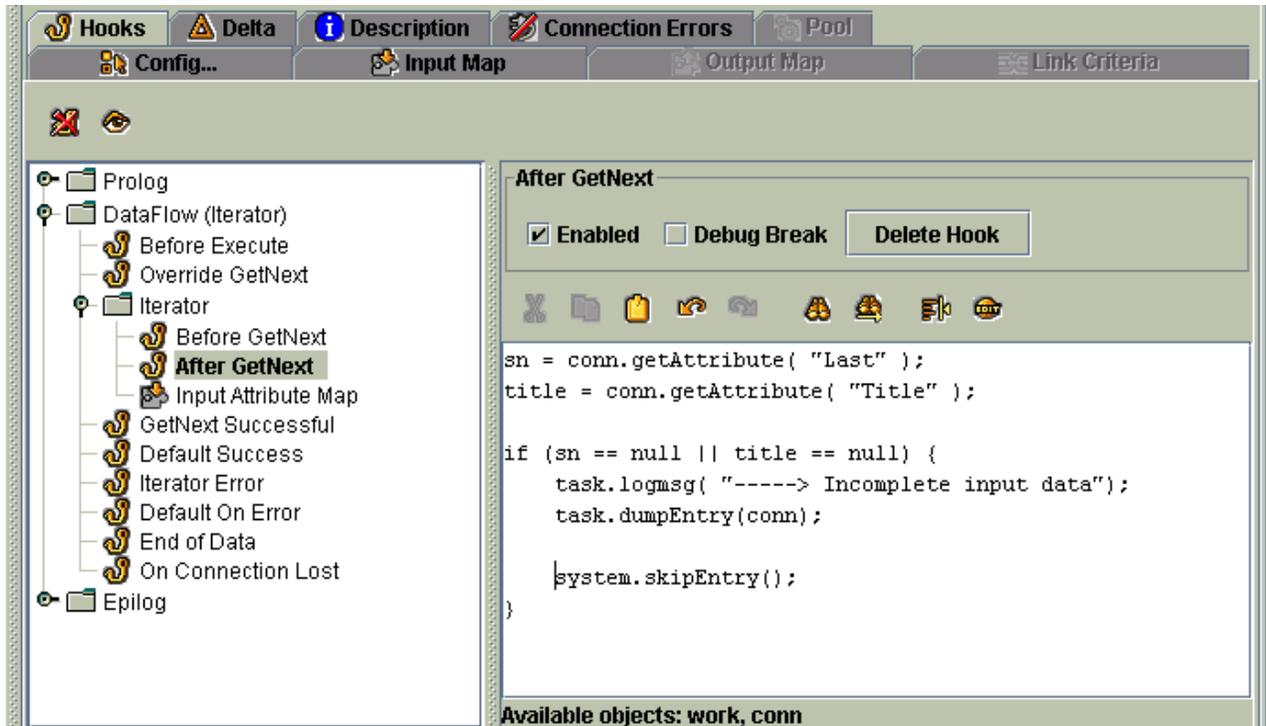
Hooks in this section are executed during each cycle of the AssemblyLine. Hooks such as **Before Execute**, **Default Error** and **Default Success** are common to all Connector modes. Most of the others are mode-specific. Since your **InputPeople** Connector is in Iterator mode, it is doing a number of GetNext operations to retrieve successive Entry objects from the input data set. Iterator mode gives you Hooks such as **Before GetNext** and **After GetNext** so that you can wrap this read operation in your own logic. All Connectors also have an **On Connection Lost** that is invoked whenever an exception occurs that TDI identifies as a connection error. This Hook is called before any Reconnect attempts are made.

After Epilog

These Hooks are started once at the end of the AssemblyLine's life-cycle, before and after the Connector closes its connection.

With the **InputPeople** Connector selected, click the **After GetNext** Hook in the list. Notice how this Hook is executed just before the Input Map is performed. This means that the data read in from the CSV file is still only available in the Connector's local **conn** object. Enter the following script to handle filtering of incomplete data:

Screen capture filename: GettingStarted-48.eps



Let's walk through this script code:

```
sn = conn.getAttribute( "Last" );
title = conn.getAttribute( "Title" );
```

The first two lines attempt to retrieve and store two attributes that should be available in the **conn** object¹⁶.

```
if (sn == null || title == null) {
```

This code checks to see if the Attributes **Last** and **Title** exist in the **conn** Entry object (e.g. if they were found in the CSV file). If not, then the `.getAttribute()` calls return `null` and the next three lines are executed (everything inside the curly braces denoting a code block).

```
task.logmsg( "-----> Incomplete input data");
```

16. If you were to script your logic in the **GetNext Successful** or **Default Success** Hooks instead, then since these Hooks come after the Input Attribute Map, your script could reference the **work** object instead.

The **task** object gives you access to AssemblyLine functions like `logmsg()`, allowing you to include messages in the AssemblyLine log.

```
task.dumpEntry(conn);
```

This AssemblyLine function displays the contents of the specified Entry object to the log. You can use it to view standard variables like **work** and **conn**, as well as any Entry objects you create yourself..

```
    system.skipEntry();  
}
```

Finally, the **system** object's `skipEntry()` function tells the AssemblyLine to stop processing the current Entry, pass control back to the start of the AssemblyLine cycle (e.g. the active *Feeds* Connector) and get the next input object.

As an extra exercise, you can take a simpler and more legible approach to solving this problem.

1. Stay out of the Hooks. Instead, insert a Branch just before your **XML_Output** Connector.
2. Add a Condition for the Branch to test if the Last Attribute does not exist (which means requires you to click **Negate** for the Condition).
3. Under this Branch, insert a Script component that displays the log message and then skips the Entry as shown in the Hook code above.
4. Disable the **After GetNext** Hook once you have your Branch logic in place.

Branches are described later in this guide, but go ahead and try this yourself now and discover how this approach not only makes your AssemblyLine easier to read, it also allows you to step through the logic using the **Step (paused)** run mode.

In keeping with the iterative, try-test-refine methodology of IBM Tivoli Directory Integrator, you are now ready to test your solution again. But before running the test, you will make a slight change to **XMLOutput** Connector's Output Map.

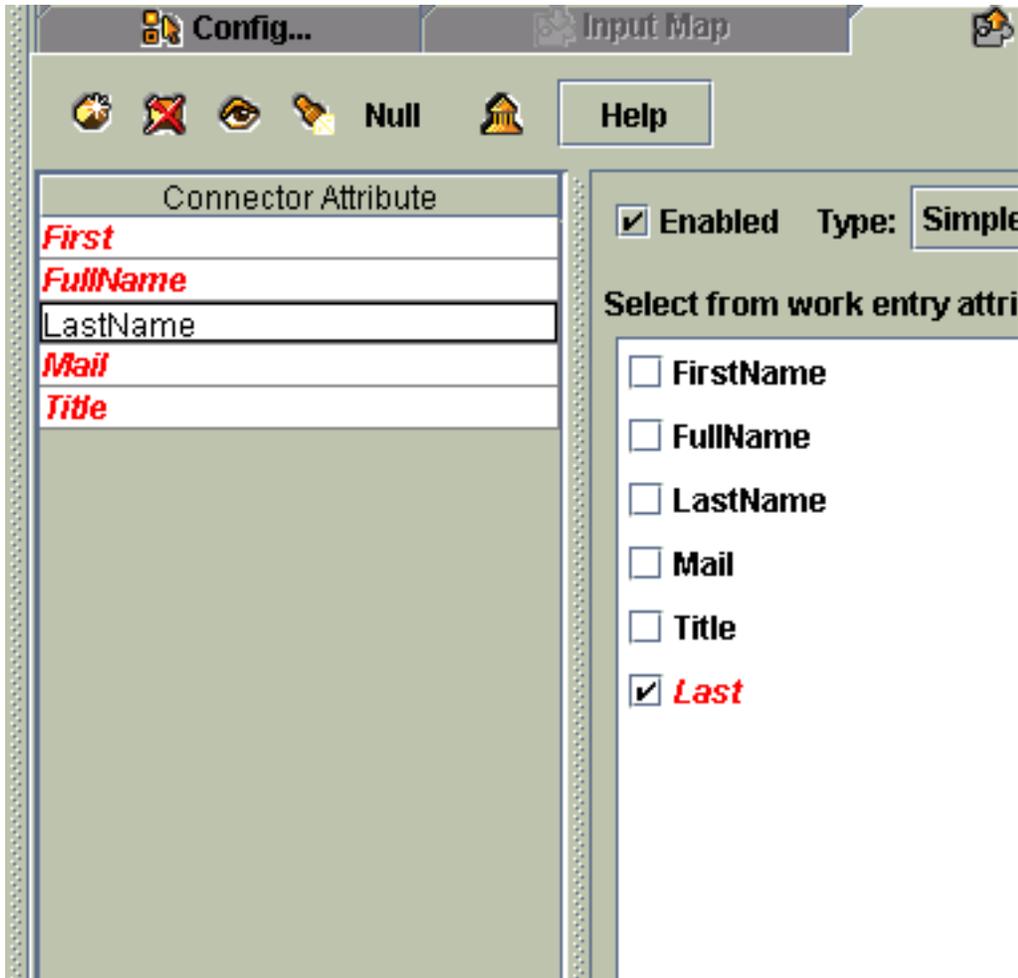
Schema conversion

In our example (see page 6), the output attributes required by DS3 have the same names as those of the input source (DS1). Imagine for a moment that the specification called for output attributes to be named *FirstName* and *LastName* instead:

```
FirstName    =DS1.First  
LastName     =DS1.Last  
FullName    =DS1.First+" "+DS1.Last  
Title       =DS1.Title  
Mail        =<compute_from_name>
```

IBM Tivoli Directory Integrator makes mapping attribute names between schemas easy, and all you need to do is update the names of these attributes directly in any Attribute Map.

So as not to affect maps and script logic that expect to find the **First** and **Last** Attributes, you will do the renaming in the Output Map of the **XMLOutput** Connector. Select this Connector now and then double-click on the attribute that you want to change in the Output Map — and just start typing.



Don't worry about this causing scripts in the AssemblyLine to fail. These names are being changed locally for the Output Map phase only. The **First** and **Last** attributes are still read in correctly, and are available throughout the AssemblyLine under their original names.

Note: Because IBM Tivoli Directory Integrator keeps focus in the field that you are entering even if you switch to a different Connector or AssemblyLine, you can be left *hanging* in edit mode for the attribute name. If you then test your AssemblyLine, your change will not be sent to the Server. To leave edit mode, either click on a different attribute in the same map, or press the Enter key so that IBM Tivoli Directory Integrator knows you are done.

Leave these changes in (even though this is not part of your original specification) and then run the AssemblyLine again. When execution completes, go back to the output browser window and click the **Refresh** button.

When the output file is visible again, confirm that **Roger** is no longer there (he used to be between **Jill** and **Gregory**). You should also see your *incomplete input data* message in the log, as well as the changes you made to two of the attribute names.

Save your Config again (Ctrl + S), and go to the next step: aggregating data from a third data source.

Adding the Join Connector

Included with the Tutorial files is a simple database of people who owe money. Use this source to include debt information in the output XML document.

Adding a new data source to our flow means inserting a third Connector in our AssemblyLine. Do this now, calling this Connector **Debtors** and choosing the **BTreeObjectDB** Connector. This Connector needs to be in **Lookup** mode since you want it to search a record that matches data brought into the AssemblyLine by your **InputPeople** Connector. You have now worked with Connectors in three different modes: Iterator, AddOnly and now Lookup. Before we continue, let's take a closer look at the various modes available for Connectors, and what these mean to your solution.

As mentioned previously, the Mode setting tells the AssemblyLine what role this Connector will play in the data flow: Will it be feeding the AssemblyLine with data, writing information, deleting it or performing lookups. Connectors can be set to one of eight possible modes, each providing specialized behavior and options for customization:

Note: It is possible to create custom Connector Modes using the AssemblyLine Connector in combination with AL Operations. For more information, see the *IBM Tivoli Directory Integrator 6.1.1: Users Guide*.

AddOnly (Flow)

This mode is for Connectors that add new information to a data source, for example, writing to files, or creating new records in a database or entries in a directory. AddOnly Connectors show up in the *Flow* part of the AssemblyLine.

Delete (Flow)

Appearing in the *Flow* section as well, Delete mode Connectors search for and then delete a specified entry. Search criteria that the Connector uses to pinpoint the entry to be deleted is specified when you set up the Connector.

Iterator (Feeds)

A Connector in Iterator mode appears in the *Feeds* section of the AssemblyLine. This mode will cause the Connector to run through the contents of its data source (or a part of it, like the result of a database or directory query) and *feed* the entries one at a time to the *Flow* section components for processing. Connectors in Iterator mode are often called **Iterators** for short. An AssemblyLine can contain more than one Iterator, and these feed the AssemblyLine exclusively in succession; for example, the second Iterator taking over once the first one reaches the end of its data set, then the third one when the second completes, and so forth.

Lookup (Flow)

Lookup Connectors reside in the *Flow* section. This mode causes the Connector to find and return entries that match the specified search criteria. This is the mode you use to aggregate, or *join* new information into your flow.

Update (Flow)

In Update mode, a Connector tries to find an entry based on search criteria settings. If the lookup succeeds, the existing entry is modified as specified in the Connector. If the lookup fails, then the Connector adds a new entry instead. Update Connectors appear in the *Flow* list.

Call/Return (Flow)

This is a special output/input mode which first sends a call package (such as a SOAP message or JMS entry) to the connected system, and then waits for a reply. These Connectors appear in the *Flow* list.

Server (Feeds)

Appearing in the *Feeds* section, Server mode Connectors listen for *events* in connected systems and then *feed* event data to the *Flow* component list. For example, you can configure the HTTP Server Connector in Server mode to monitor a desired port for incoming browser requests, or the LDAP Server Connector to accept and process LDAP requests.

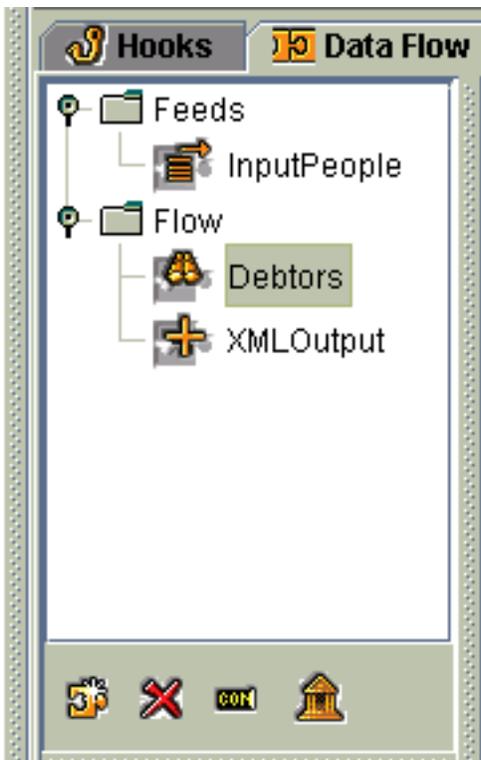
Delta (Flow)

Connectors in Delta mode are *Flow* components, and provide functionality for applying changes detected by an appropriate Change Detection Connector in the *Feeds* section. Delta mode requires the Entry and its contents to be tagged with special change operation codes, enabling it to perform the delta update (add, modify or delete) in the most efficient fashion.

Returning to our example, when you added your new Connector, IBM Tivoli Directory Integrator placed it at the end of the AssemblyLine. But this won't work because you need to do the *join* just after the initial input (iteration), but before output to XML.

To fix this, simply select and drag the Connector to the top of the *Flow* section.

Screen capture filename: GettingStarted-52.eps

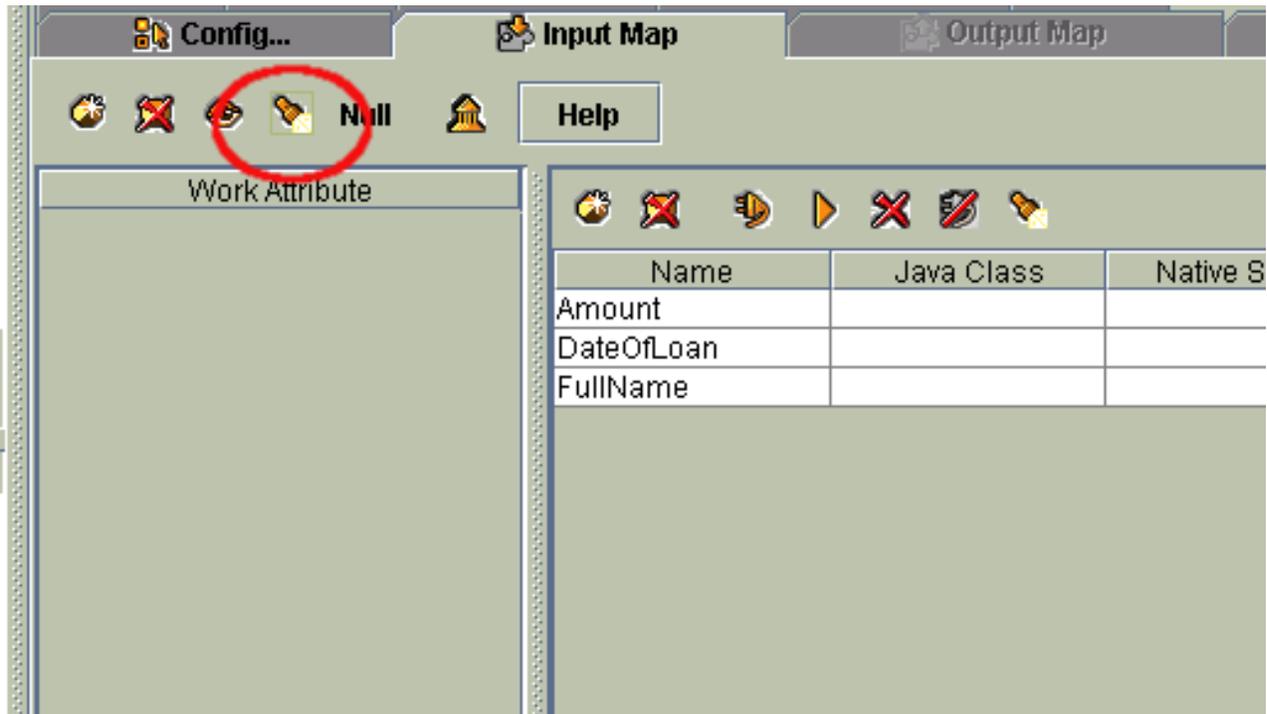


Now enter the path name of the BtreeObjectDB data file. Note that the database file itself is called `Debtors.dat` and should be located in the `examples/Tutorial`

sub-directory. In the **Key Attribute Name** field you need to specify the name of the attribute that uniquely identifies these records. For this tutorial database, the Key Attribute is **FullName**¹⁷.

Test your parameter settings like you do with any Connector by selecting the **Input Map** tab. However, instead of using the **Connect** and **Read next...** buttons, try the **Quick Discovery** button in the Attribute Map button bar.

Screen capture filename: GettingStarted-54.eps



This convenient button makes the Connector connect to its source, do a single *GetNext* on the data set and then examine the returned Entry. The attributes found here are displayed in the Connector Schema list and are ready for mapping. Now although this does not discover all the attributes defined in the schema of the data source (you need to use the **Discover Schema** button found at the top of the Connector Schema list for this) it is enough for you to continue your work.

Set up the Input Map by dragging **Amount** and **DateOfLoan** from the Connector Schema list into the map. You should see these two attributes appear in the Work Entry display, indicating that they are now available in the AssemblyLine.

Note: As you'll see in the following section, you will use the newly discovered **FullName** attribute to set up the search criteria for the Lookup, even though you did not include it in the Input Map. An attribute does not need to appear in the Input Map in order for you to use it for searching — just as long as it's available in the data source.

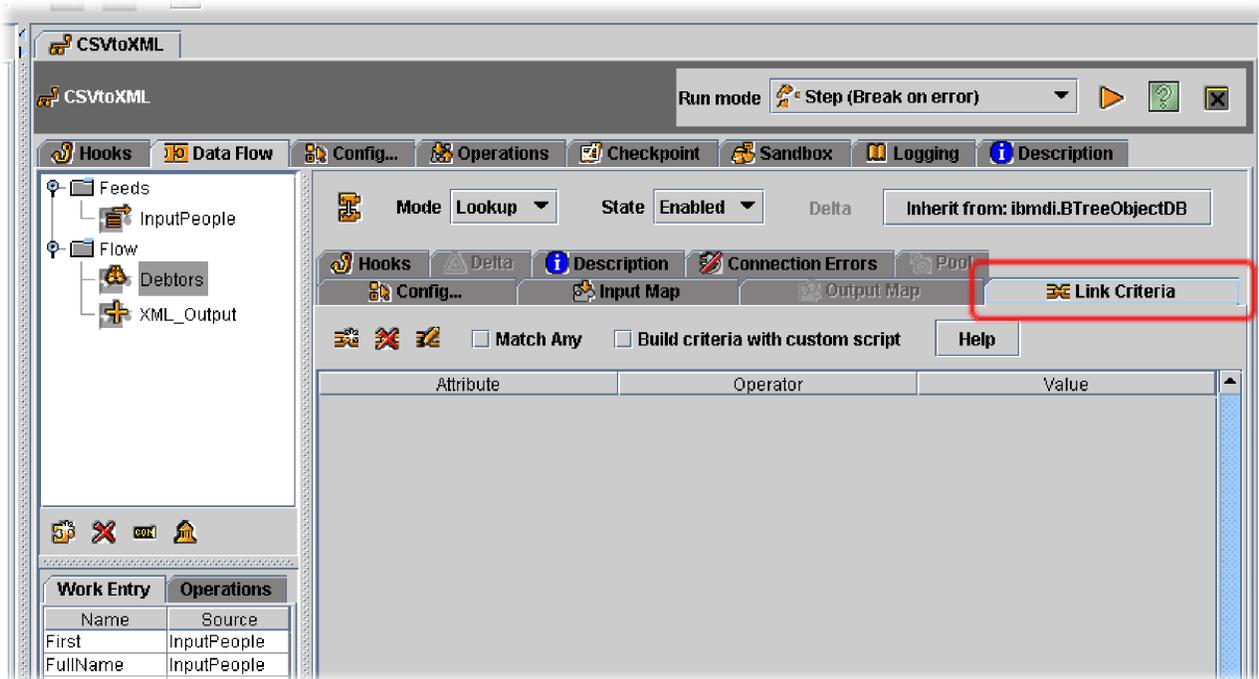
¹⁷. And now you know why you needed to construct this attribute in the Input Map of your Iterator

Setting up Link Criteria

Because it's in Lookup mode, the new **Debtors** Connector is searching for specific entries in its data source, trying to find a match for the entry that is already inside the AssemblyLine. Exactly how this match is made is determined by search rules called the Connector's **Link Criteria**.

Since the **Debtors** Connector is in a mode that requires it to find matching entries in its data source, the tab called **Link Criteria** is enabled.

Screen capture filename: GettingStarted-55.eps



Selecting this tab brings up the Link Criteria display where you can create one or more simple *Link Criteria* matching rules. These rules are AND'ed together by the Connector to make the system-specific call; or if you select the **Match Any** checkbox, then a logical OR is used instead. Since each Connector knows how to translate your Link Criteria to the relevant syntax for the underlying data source, your solution is kept technology independent¹⁸.

Click the **Add new Link Criteria** button in the Link Criteria toolbar to configure a new search rule:

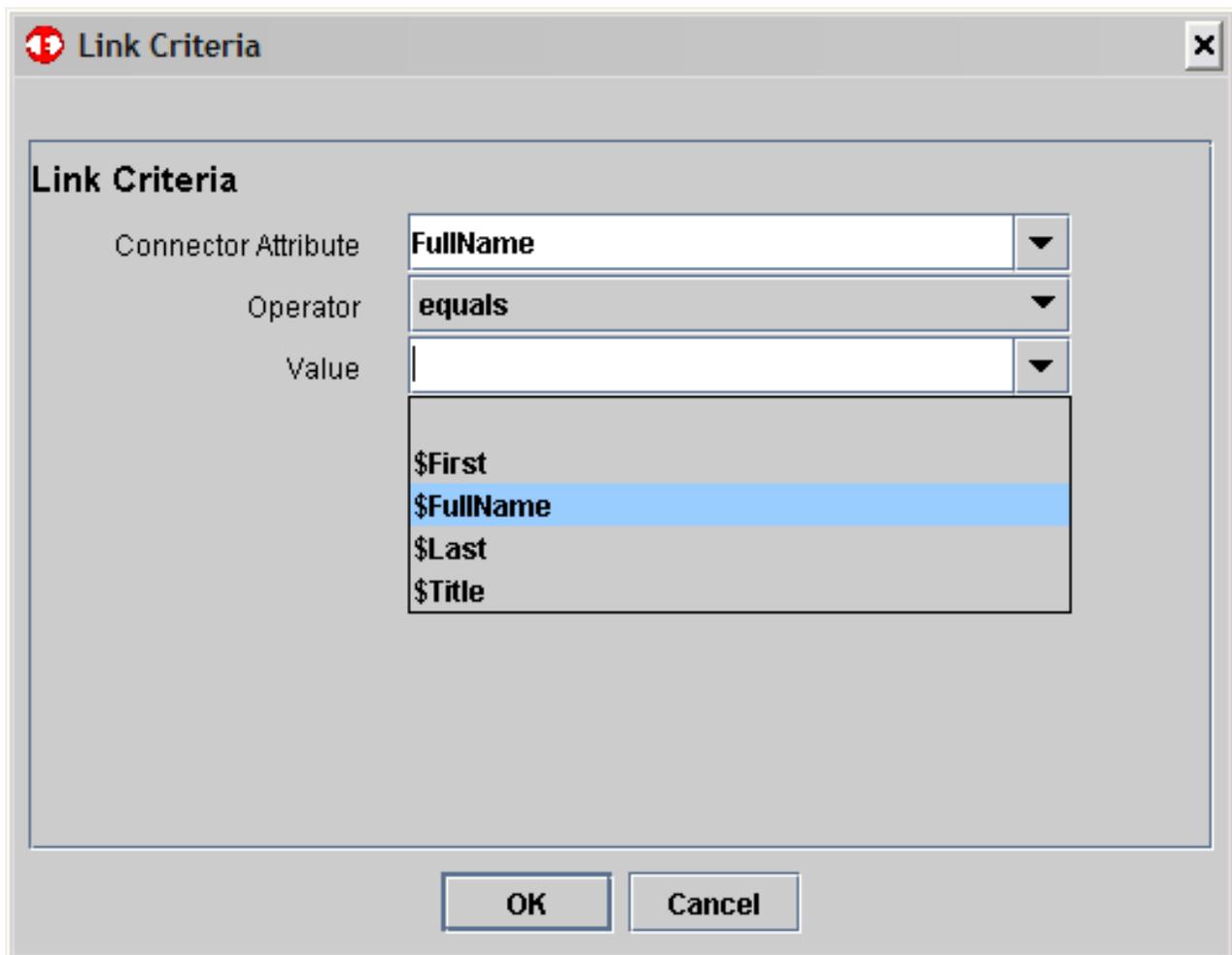
18. Remember the Advanced Mapping feature for scripting an Attribute Map yourself? This same principle applies here. By selecting the **Build criteria with custom script** checkbox, you get a Script Editor window where you can write code to create the data source-specific lookup call. Here you must set `ret.filter` to the search phrase you want the Connector to use. For example, this would be the WHERE clause of an SQL SELECT statement for a JDBC Connector, or an LDAP search filter if you are working with a directory. Using scripted criteria allows you to build complex queries. However, the trade-off is that your solution becomes more tightly bound to the underlying stores and technologies.

Screen capture filename: GettingStarted-56.eps



When the Link Criteria dialog box appears, choose **FullName** for the **Attribute** parameter, selecting it from the schema that IBM Tivoli Directory Integrator discovered in the data source. Then select the **equals** comparison **Operator**. Finally, choose **\$FullName** from the **Value** menu.

Screen capture filename: GettingStarted-57.eps



Notice how the **Value** menu only shows you those Attributes that are present in the flow *before* this Connector is reached. Those Attributes brought into the AssemblyLine by the **Debtors** Connector are not available yet, and cannot be used to set up Link Criteria at this point.

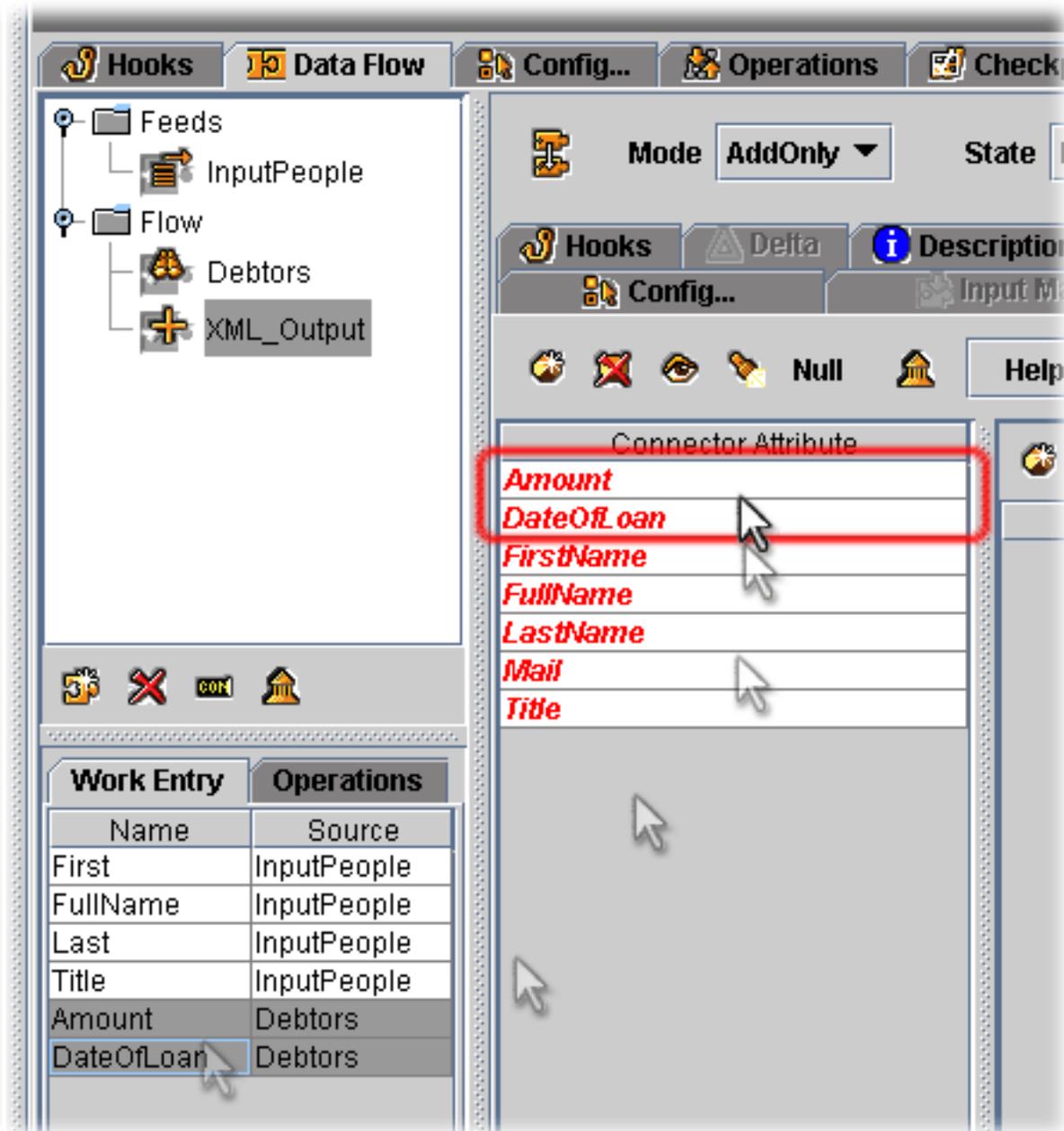
Click **OK** when you are done.

Note: The dollar sign (\$) character in front of the **FullName** attribute in the AssemblyLine instructs IBM Tivoli Directory Integrator to retrieve the *first*

value of the named attribute to use in building the Link Criteria. If you want to match any one of the values of a multi-value attribute, you can use the *at* symbol (@) instead. To search for a literal value, simply enter the desired text in the **Value** parameter directly.

Now the join is ready and your last task is to tell the output Connector to include the new Attributes (mapped in by the **Debtors** Connector) in the XML output. Do this by selecting the **XMLOutput** Connector and its **Output Map** tab. From the Work Entry box, drag the new **Amount** and **DateOfLoan** attributes onto the map.

Screen capture filename: GettingStarted-59.eps



Save your work and run the AssemblyLine again.

What happened now? The AssemblyLine crashed with the following log output:

```

...
15:07:53 BEGIN Iteration
15:07:54 [Debtors] Lookup
java.lang.Exception: [Debtors] Entry not found
  at com.ibm.di.server.Log.exception(Unknown Source)
  at com.ibm.di.server.AssemblyLineComponent.lookup(Unknown Source)
  at com.ibm.di.server.AssemblyLine.msExecuteNextConnector(Unknown Source)
  at com.ibm.di.server.AssemblyLine.executeMainStep(Unknown Source)
  at com.ibm.di.server.AssemblyLine.executeMainLoop(Unknown Source)
  at com.ibm.di.server.AssemblyLine.executeAL(Unknown Source)
  at com.ibm.di.server.AssemblyLine.run(Unknown Source)
15:07:54 Error in: NextConnectorOperation: java.lang.Exception: [Debtors] Entry not found
java.lang.Exception: [Debtors] Entry not found
  at com.ibm.di.server.Log.exception(Unknown Source)
  at com.ibm.di.server.AssemblyLineComponent.lookup(Unknown Source)
  at com.ibm.di.server.AssemblyLine.msExecuteNextConnector(Unknown Source)
  at com.ibm.di.server.AssemblyLine.executeMainStep(Unknown Source)
  at com.ibm.di.server.AssemblyLine.executeMainLoop(Unknown Source)
  at com.ibm.di.server.AssemblyLine.executeAL(Unknown Source)
  at com.ibm.di.server.AssemblyLine.run(Unknown Source)
15:07:54 BEGIN Connector Statistics
15:07:54 [InputPeople] Get:1
15:07:54 [Debtors] Errors:1
15:07:54 [XMLOutput] Not used
15:07:54 Total: Get:1, Errors:2
15:07:54 END Connector Statistics
15:07:54 failed with error: [Debtors] Entry not found
15:07:54 AssemblyLine AssemblyLines/CSVtoXML failed with error: [Debtors] Entry not found

```

Whenever you get an error that is not handled by your solution, the system gives you an exception dump similar to the one above. If you examine the messages at the top of the dump you can see that IBM Tivoli Directory Integrator is telling you exactly where the error occurred:

```

...
15:07:54 [Debtors] Lookup
...

```

So your **Debtors** Connector failed during the lookup operation. Furthermore, the next line details the type of exception that occurred:

```

...
java.lang.Exception: [Debtors] Entry not found
...

```

This tells you that your AssemblyLine failed because you have not accounted for the situation where the **Debtor** Lookup Connector fails to find a match its database, i.e. that these people do not owe money. Fortunately, IBM Tivoli Directory Integrator has a number of ways to deal with this situation. In this case, you can program a *Hook* to remedy the problem.

The logic you implement to deal with the failed lookup can handle the exception in a number of ways:

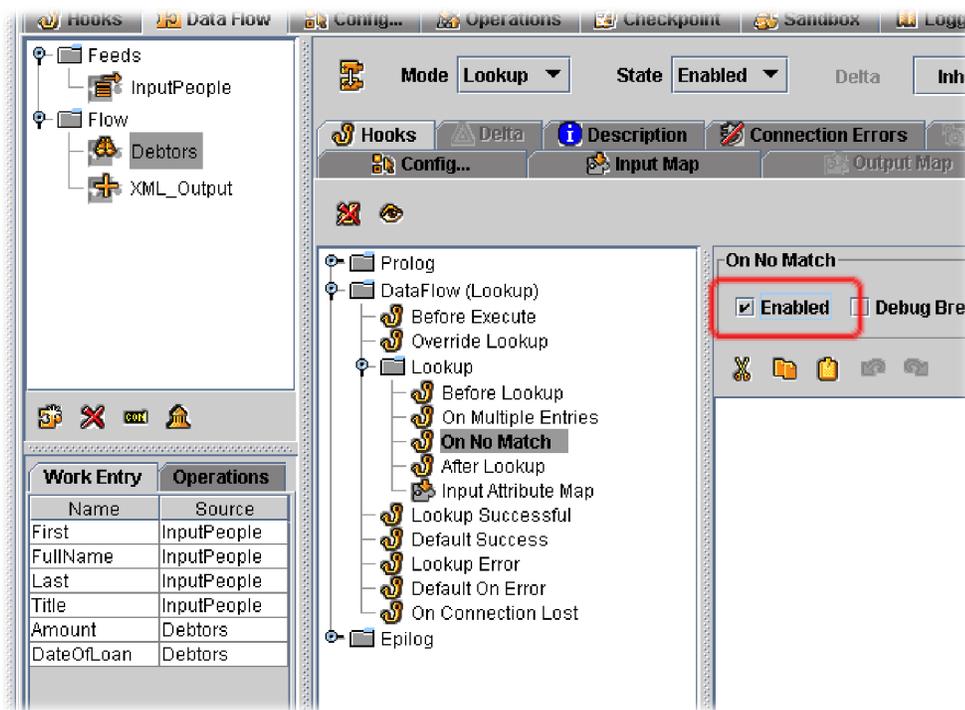
- You can choose to ignore the fact that the entry sometimes does not exist in the Debtors.dat database. In that case, you instruct the program to ignore the exception and move on to the output Connector. Since no matching entry was found, the Connector cannot map in the Attributes you specified. As a result, your output contains all entries, but only some with debt information.

- Or you might want to include only people with debts in your XML document, in which case you instruct the system to skip any Entries that have no matching data in Debtors.

If you wanted to simply ignore the error, then all you have to do is enable the Hook called **On No Match** in the **Debtors** Connector:

1. Select the **Debtors** Connector and click its **Hooks** tab.
2. Click on the **On No Match** Hook in the tree-view that is presented here. A Script Editor window is displayed.
3. Check **Enabled** , save your work and run the AssemblyLine again.

Screen capture filename: GettingStarted-65.eps



By enabling the Hook, you are telling IBM Tivoli Directory Integrator not to do anything if no matching data is found during the Lookup.

When you run the AssemblyLine again it does not crash, and the log output looks like this:

```
15:27:20 Loading configuration from <stdin>
15:27:20 Starting AssemblyLine AssemblyLines/CSVtoXML
15:27:21 AssemblyLine AssemblyLines/CSVtoXML started
15:27:22 [Debtors] Enabled hook script for lookup_nomatch is empty
15:27:22 [InputPeople] Using first input line for column names
15:27:22 BEGIN Iteration
15:27:22 -----> Incomplete input data
15:27:22 *** Begin Entry Dump
15:27:22   Operation: generic
15:27:22   [Attributes]
15:27:22     First (replace):   'Roger'
15:27:22 *** End Entry Dump
15:27:22 END Iteration
15:27:22 BEGIN Connector Statistics
15:27:22   [InputPeople] Get:6, Skip:1
15:27:22   [Debtors] Lookup:4
15:27:22   [XMLOutput] Add:6
15:27:22 Total: Get:6, Lookup:4, Add:6, Skip:1
15:27:22 END Connector Statistics
15:27:22 terminated successfully (0 errors)
15:27:22 AssemblyLine AssemblyLines/CSVtoXML terminated successfully
15:27:23 Server terminates because only main thread left
```

Here you can see the messages and entry dump that you coded into your filtering script (in the **After GetNext Hook** of **InputPeople**), as well as the message that no errors were encountered this time.

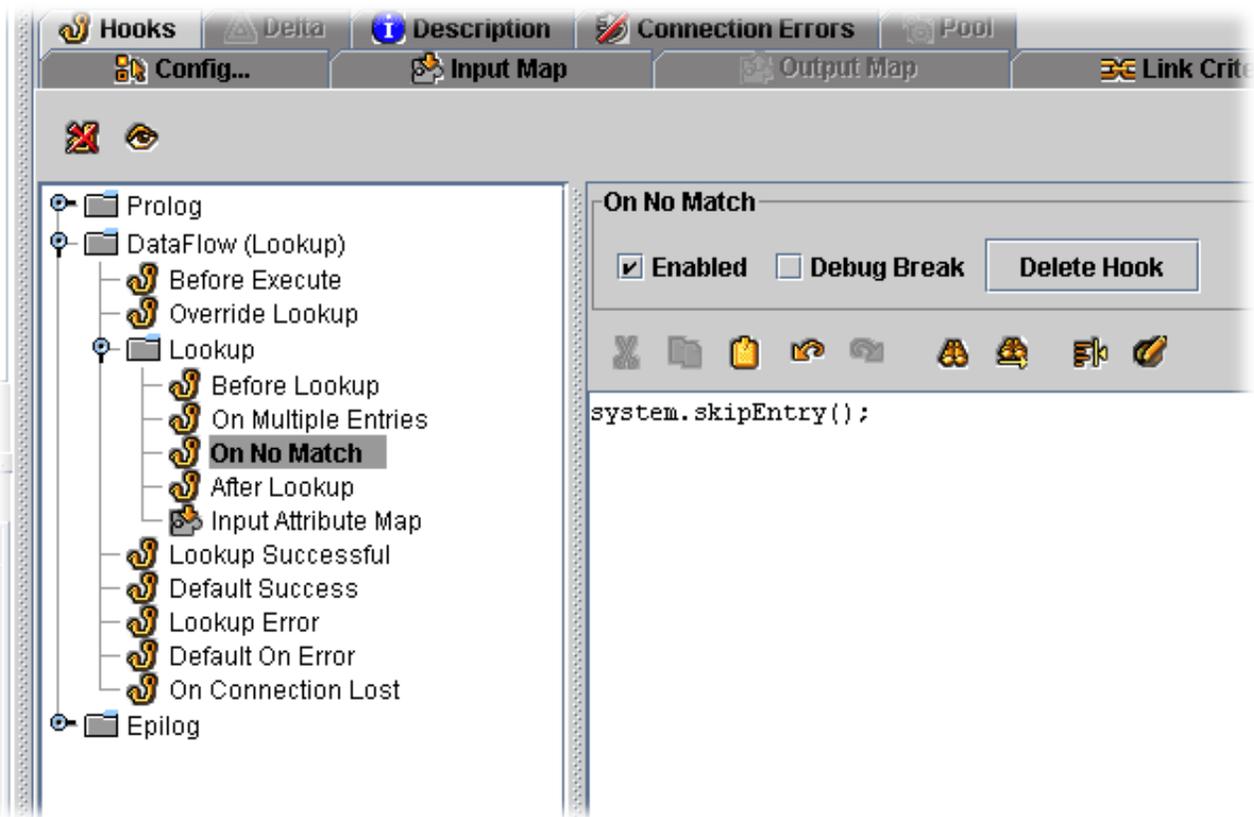
Your XML output file shows a couple of important changes as well:

- The two new **Debtors** fields that were included are visible in the output.
- The number of attributes per XML entry became variable, since the AssemblyLine did not find debt information for all of them.

Note: You may have noticed that the XML document tags do not appear in the same order as they are listed in the Output Map. Attributes Maps are sorted alphabetically, and this ordering will generally have no effect on the order of attributes in your output.

But let's say that you are only interested in those people that do owe money. Then you will want to implement the second option outlined above, effectively filtering Entries for people with no matching Debtors data. To do this, add the following snippet of script to the **On No Match Hook** in the **Debtors Connector**:

```
system.skipEntry();
```



As you saw before, the `system.skipEntry()` command tells your `AssemblyLine` to skip the current `Entry`, go back to the top and let your `Iterator` grab the next one. So now your completed solution is doing two types of data *filtering*: a) your **PeopleInput** `Iterator` is skipping incomplete records in the CSV file, and b) the **Debtors** `Lookup Connector` is passing only debtors to output.

Now that you've gotten a grip on building, testing and running an integration solution using IBM Tivoli Directory Integrator, it's time to look at how you wire event-awareness into them.

Event-driven Integration

Up to this point, your solution has been batch-oriented; you have been launching your `AssemblyLine` manually from within the `Config Editor`. You can also launch your `AssemblyLine` from the command line:

```
ibmdisrv -c examples/Tutorial/Tutorial1.xml -r CSVtoXML
```

Launching from the command line allows you to use batch mechanisms to fire off your `AssemblyLines` on a scheduled basis, or at the request of external applications.

However, IBM Tivoli Directory Integrator provides you with the tools to build event-awareness directly into your solutions¹⁹, allowing you to take actions at timed intervals based on changes in data sources, messages coming in via mail, IP packets or MQ.

Event handling can be done in a number of ways:

Connectors in Iterator Mode

Some Connectors allow you to configure timeout parameters for Iterator Mode. One example is the FileSystem Connector, which can be set up to read through a file to the end and then wait for new information to appear – a so-called *tail read*.

Other Connectors, like those for RDBS Change detection and LDAP Changelog, work in a similar way. These Connectors allow you to build AssemblyLines that run continuously, waiting for new changes to appear in the connected system. There is also a Timer Connector that runs in Iterator Mode and can be configured to drive your AssemblyLine at timed intervals according to a scheduling parameters.

Connectors in Server Mode

There are a few specialized Connectors, like the HTTP Server Connector and the LDAP Server Connector, that use the Connector Server Mode. These components allow you to build solutions that process incoming requests from external clients.

For example, the HTTP Server Connector can be configured to listen to a specific IP port waiting for connection requests. When a client makes a request, such as HTTP GET or POST, the Connector creates a copy of itself in Iterator Mode in order to retrieve the data and pass it to the **Flow** section of the AssemblyLine. The Server Mode Connector itself immediately returns to listening mode to catch new events²⁰. In addition, the Connector automatically clones a new copy of itself for each client that makes a connection.

Working in a similar fashion, the LDAP Server Connector enables your AssemblyLines to accept incoming LDAP requests, enabling you to build solutions that provide virtual directory services.

Notifications and properties

TDI has components that can subscribe to TDI notification events, just as there are components (and script calls) for sending these events.

In the following example, you will use the same techniques that you've learned in the previous exercises, except that this time when you run the AssemblyLine, it will not stop. The AssemblyLine will wait for an event to trigger it. When an event occurs, your AssemblyLine will service the event and go back to listening again. This is what a Server Mode Connector does.²¹.

19. IBM Tivoli Directory Integrator also boasts a rich API allowing you to configure and manage a running Server without ever even starting the Config Editor. There is also a command line interface program (tdisrvctl.bat under the bin sub-directory) that lets you connect to any running TDI Server. Using the CLI you can load and control solutions from the command prompt, from batch-files and shell scripts.

20. Note that you can define an AssemblyLine pool in the AssemblyLine **Config** tab. This is actually a pool of the AssemblyLine **Flow** list so when you have a lot of incoming traffic, the Server Mode Connector will have enough pre-initialized and "hot" AssemblyLines to deal with these events

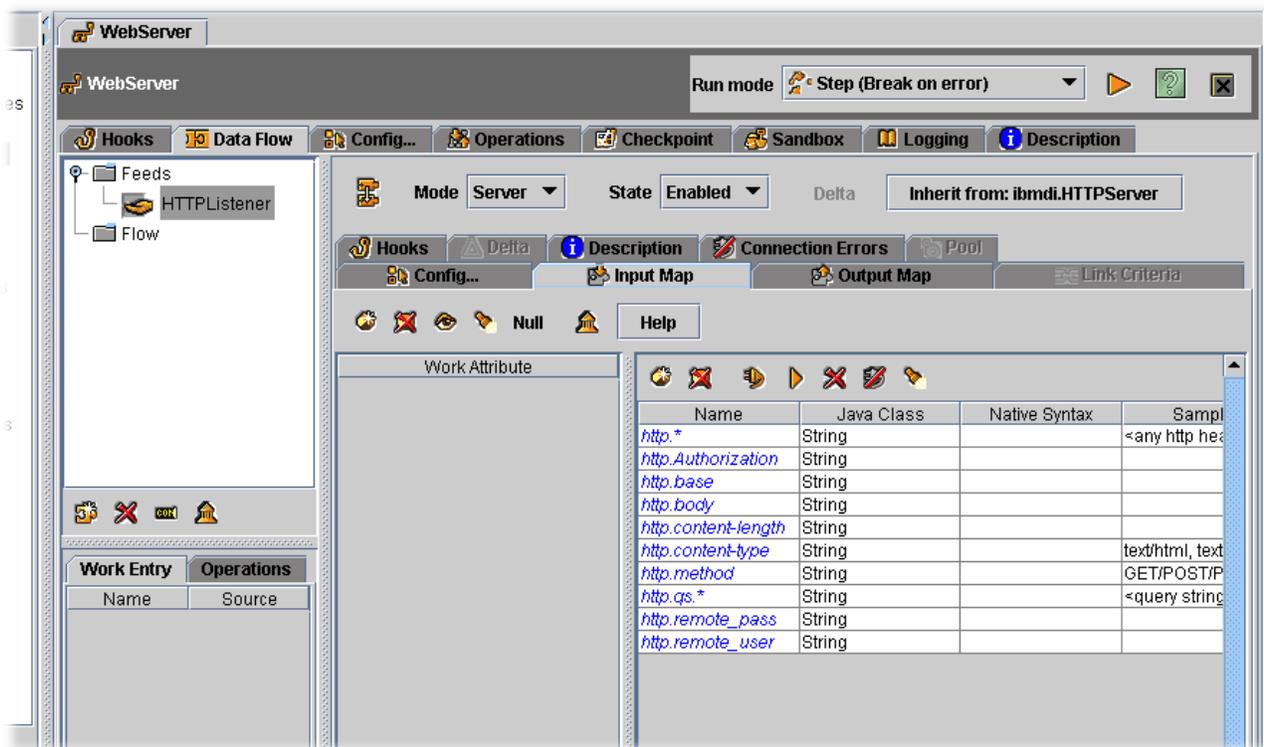
21. You can think of Server Mode as being three modes rolled into one: First you have the actual Server Mode operation of connecting to a resource and waiting for a client to connect; And then accepting (or rejecting) these incoming connections. If the client call is accepted, the Server Mode Connector creates a copy of itself in Iterator Mode.

In this example you'll use a Server Mode Connector in your Feeds section. Server Mode tells the Connector to bind to a resource (like an IP port) and wait for incoming connections. The one you'll use is called the **HTTP Server Connector**, and it provides the framework for a Web server at the configured IP port.

Start by creating a new AssemblyLine and naming it **WebServer**. Then add a Connector, selecting the **HTTP Server Connector** type and calling it **HTTPListener**. The only mode this Connector supports is **Server**.

The Configuration of this Connector is simple – just leave the default settings as they are. The Web server will be available on port 80. The next step is to discover what Attributes are available for mapping. Do this by selecting the **Input Map** tab.

Screen capture filename: GettingStarted-66.eps



The schema for this Connector is already discovered. Like the other Server Mode Connectors, it has a hard-coded Schema²². This is because it's difficult to connect and discover the schema of a client – you have to have one connect to you first. So for convenience, these components show you what to expect.

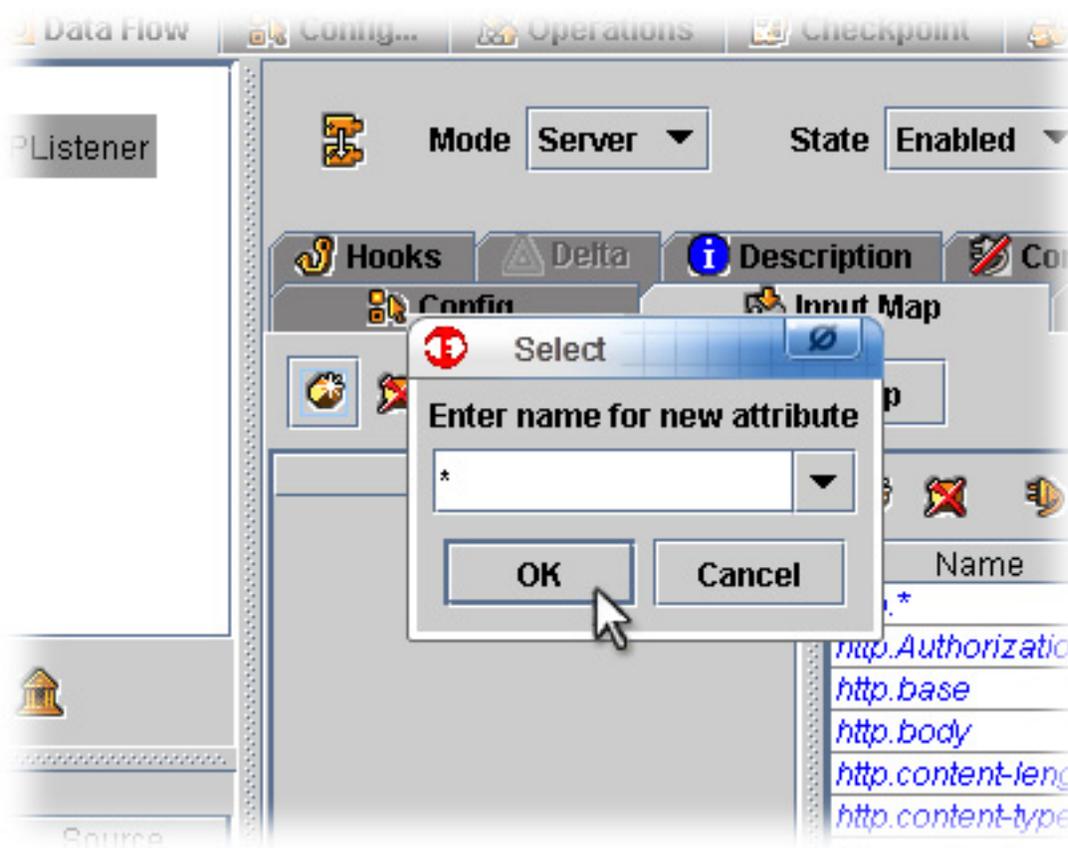
This Iterator is then attached to the AssemblyLine Flow section (making this AssemblyLine look a lot like the one you were just working on: an Iterator feeding data to Flow components). This leaves the Server Mode Connector free to go back to listening again while the newly constructed AssemblyLine handles the call. Note that the Server Mode Connector has also added a Reply Mode Connector to the end of this AssemblyLine – after the Flow section.

So, when Flow processing is complete, the Reply Mode Connector uses the Work Entry that just left the Flow section to send a response back to the client. If you look at the Hooks of a Server Mode Connector, you will see all three: Server, Iterator and Reply. Furthermore, when you use a Server Mode Connector, you can configure your AssemblyLine to use a pool of AssemblyLine Flow sections, allowing your web server solution to handle more traffic.

22. Notice the strangely named Attributes, like "http.*", in the Connector Schema. When you use the wildcard (*) in the Input Map, then you also see these same names appear in the Work Entry. This does not mean that you'll be getting an Attribute called "http.*". Rather, it indicates that you can expect to receive a number of Attributes with names starting with "http.". Remember that the CE is a design-time tool, so sometimes you only get guidelines for what will happen at run-time.

Select all the attributes by creating an attribute named * (the asterisk) in the Input Map.

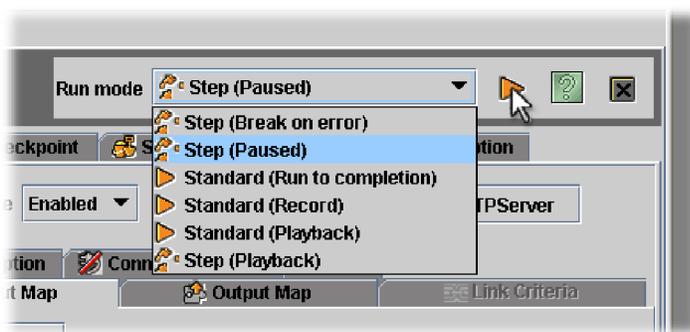
Screen capture filename: AddWildcard.eps



All Connector Schema Attributes are mapped into the Work Entry. Now go to the Output Map and add a wildcard attribute there as well. Otherwise, nothing will be mapped out into the reply message that will be sent to the client.

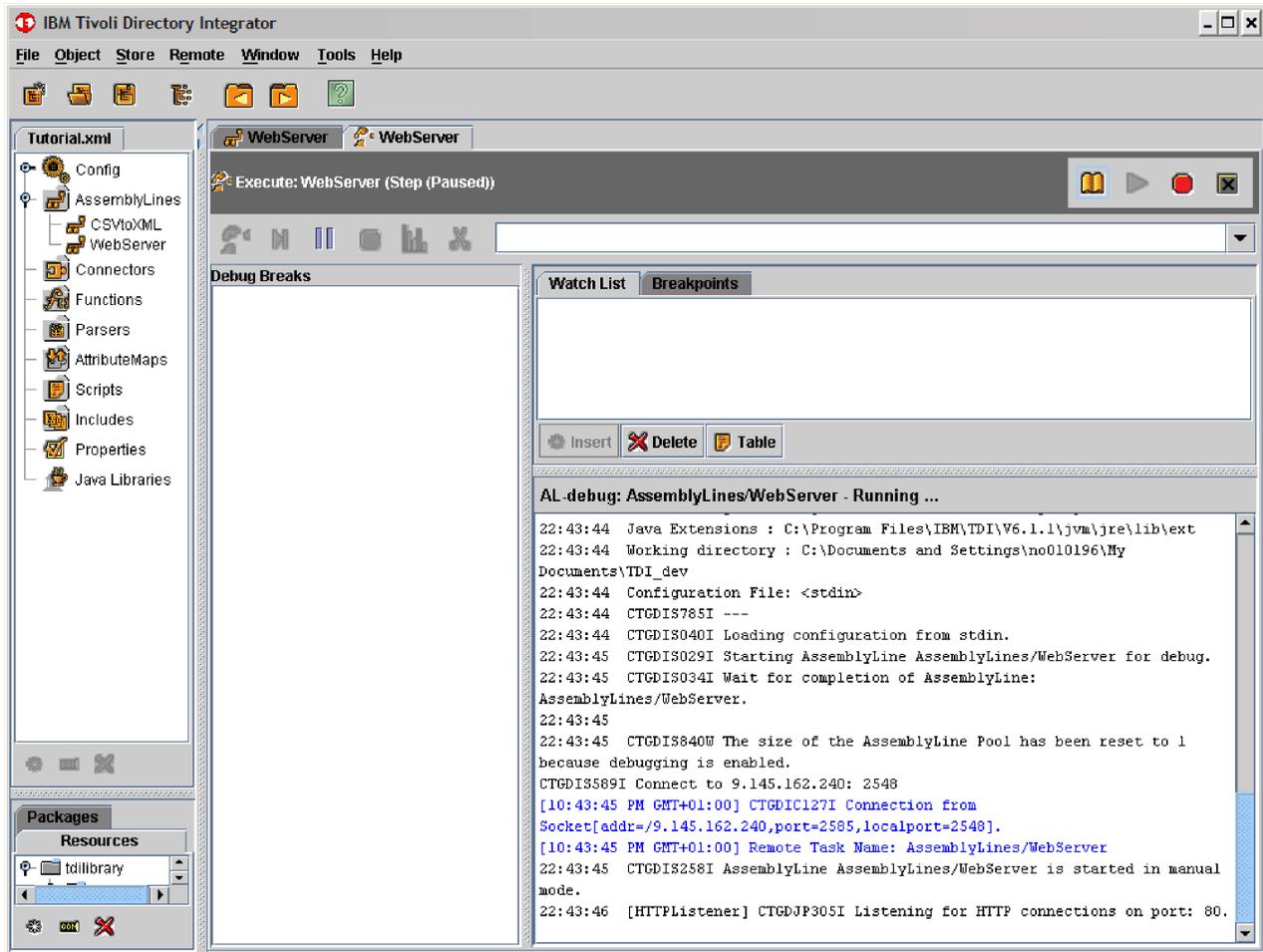
The next step is to check that the framework for your Web server is working. Do this now by clicking the **Debug** button.

Screen capture filename: rundebugger.eps



This starts your AssemblyLine just like clicking **Run** , except that your AssemblyLine is running in interactive mode. Instead of the usual log output display, the Debugger window opens instead.

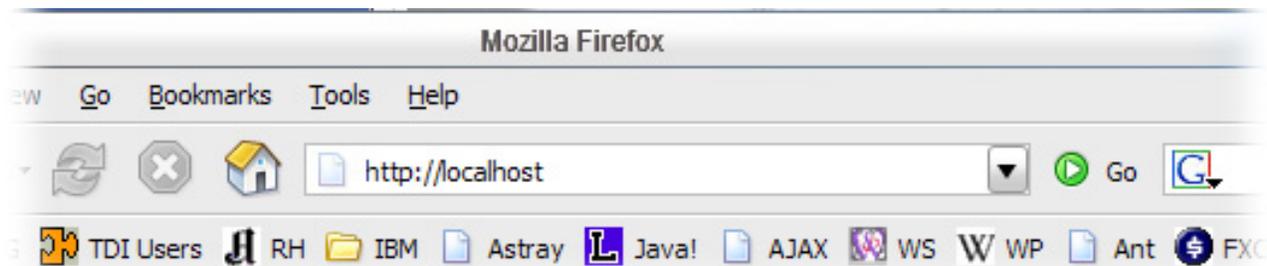
Screen capture filename: Debugger1.eps



You still get the log output in the lower right-hand window, but this now also includes debug messages in blue – both those coming from the TDI, as well those that you send yourself with scripted calls to “task.debugMsg()”.

At this point, your Server Mode Connector is listening to port 80. Nothing more will happen until you connect to it from a browser by entering the address of the socket being listened to. Open a Web browser and enter **http://localhost** in the in the address field.

Screen capture filename: DialupBrowser.eps

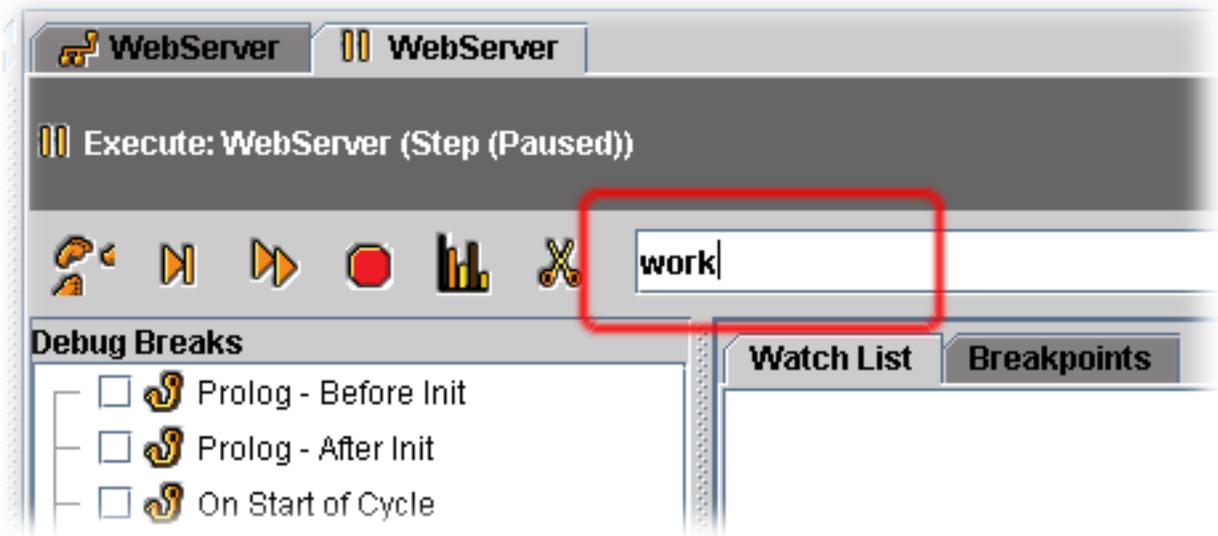


Your AssemblyLine will begin to process this null request, and you are given control. Now you can use the **Step Into** button to walk through the processing. (The **Step Into** button is the button with footprints on it.)

The first thing you will see is that your Iterator goes through the Prolog Hooks, even though this component is already initialized. Keep stepping until you get to the **GetNext Successful** Hook.

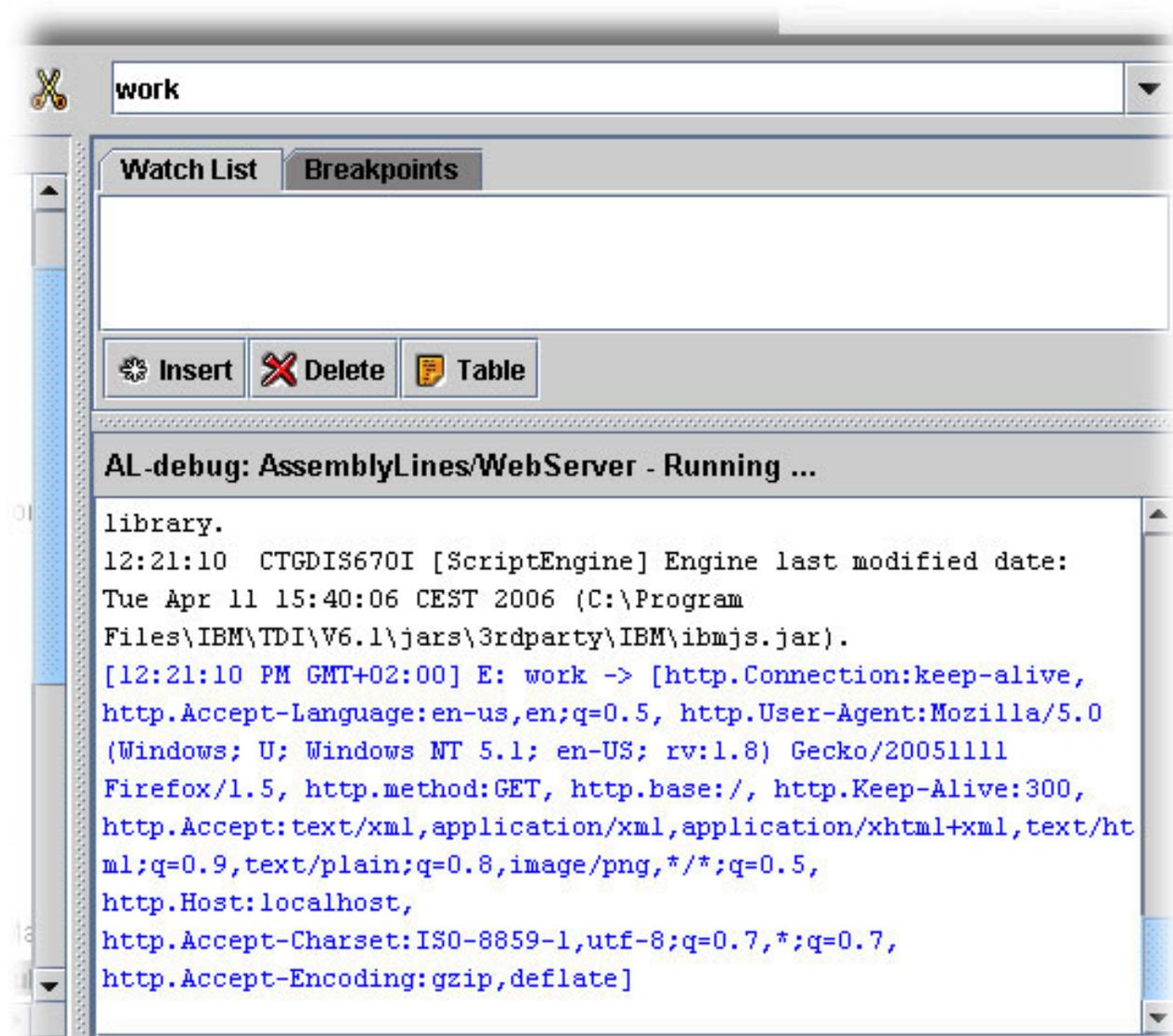
By now, your Iterator has read in the client data and marshalled it to Attributes that are first stored in the conn Entry and then copied by the Input Map into the Work Entry. Test this now by typing **work** in the **Evaluate command** field.

Screen capture filename: GettingStarted-69b.eps



When you press **Enter**, this snippet of script gets evaluated.²³ and you will see the Work Entry displayed as debug output in the log window.

23. Even a variable by itself is considered a script statement. The result of this statement is that the contents of the object is displayed; in the above example this is the string representation of the Work Entry.



Because the output is not easy to read, press **Ctrl+A** to select “work” and replace it with “task.dumpEntry(work)”. When you press **Enter** this scripted function call actually writes messages to the log, just like it does when run from script inside your solution ²⁴, so you should now see an Entry dump in the output window.

24. This is what makes the Debugger so powerful: you can inspect and change any data or configuration settings dynamically by calling any of the functions that you can do from script inside the AssemblyLine.

```

AL-debug: AssemblyLines/WebServer - Running ...
12:25:52 CTGDISO03I *** Start dumping Entry
12:25:52 Operation: generic
[12:25:52 PM GMT+02:00] E: task.dumpEntry(work) -> null
12:25:52 Entry attributes:
12:25:52 http.Accept-Language (replace):
'en-us,en;q=0.5'
12:25:52 http.Connection (replace): 'keep-alive'
12:25:52 http.User-Agent (replace): 'Mozilla/5.0
(Windows; U; Windows NT 5.1; en-US; rv:1.8) Gecko/20051111
Firefox/1.5'
12:25:52 http.method (replace): 'GET'
12:25:52 http.base (replace): '/'
12:25:52 http.Keep-Alive (replace): '300'
12:25:52 http.Accept (replace):
'text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,te
xt/plain;q=0.8,image/png,*/*;q=0.5'
12:25:52 http.Host (replace): 'localhost'
12:25:52 http.Accept-Charset (replace):
'ISO-8859-1,utf-8;q=0.7,*;q=0.7'
12:25:52 http.Accept-Encoding (replace):
'gzip,deflate'
12:25:52 CTGDISO04I *** Finished dumping Entry

```

The Server Mode Connector is working, so stop the Debugger and we'll keep building.

The first thing your AssemblyLine needs is the logic to return a web page. Implement this by adding a Script component and calling it "ReturnWebPage". Then enter the following script into this SC:

```

// First we grab the base from the HTTP property. Then we
// set up a variable to point to the Tutorial sub-directory
// -- you may have to edit this for your installation
//
var base = work.getString("http.base");
var path = "C:/Program Files/IBM/TDI/V6.1/examples/Tutorial/";

if (base == null) // Just in case :)
    base = new java.lang.String("/");

// Since getString() returns a java.lang.String, you can use the
// Java String .endsWith() function to check the extension of the
// file being requested by the client. You then set the properties
// for the return HTTP message that the Server Mode Connector
// will pass back to the browser in its reply.
//

```

```

if ( base.endsWith(".gif") )
    work.setAttribute ("http.content-type", "image/gif")
else if ( base.endsWith(".jpg") || base.endsWith(".jpeg") )
    work.setAttribute("http.content-type", "image/jpeg")
else
    work.setAttribute ("http.content-type","text/html");

// If this is a root or null request, add the name of the main web
// page ("index.html"). Note: Here we could also look for "..", to
// see if someone was trying to hack the file system.
//
if ( base == "/" )
    base = "/index.html";

// Now create a new Java File object with the path variable
// prepared above and the name of requested file.
//
var file = new java.io.File(path + base);

// Here you write the name of the file to the log. The
// java.lang.File object knows how to turn itself into a String
// when we use it like one.
//
task.logmsg ("File request: " + file );

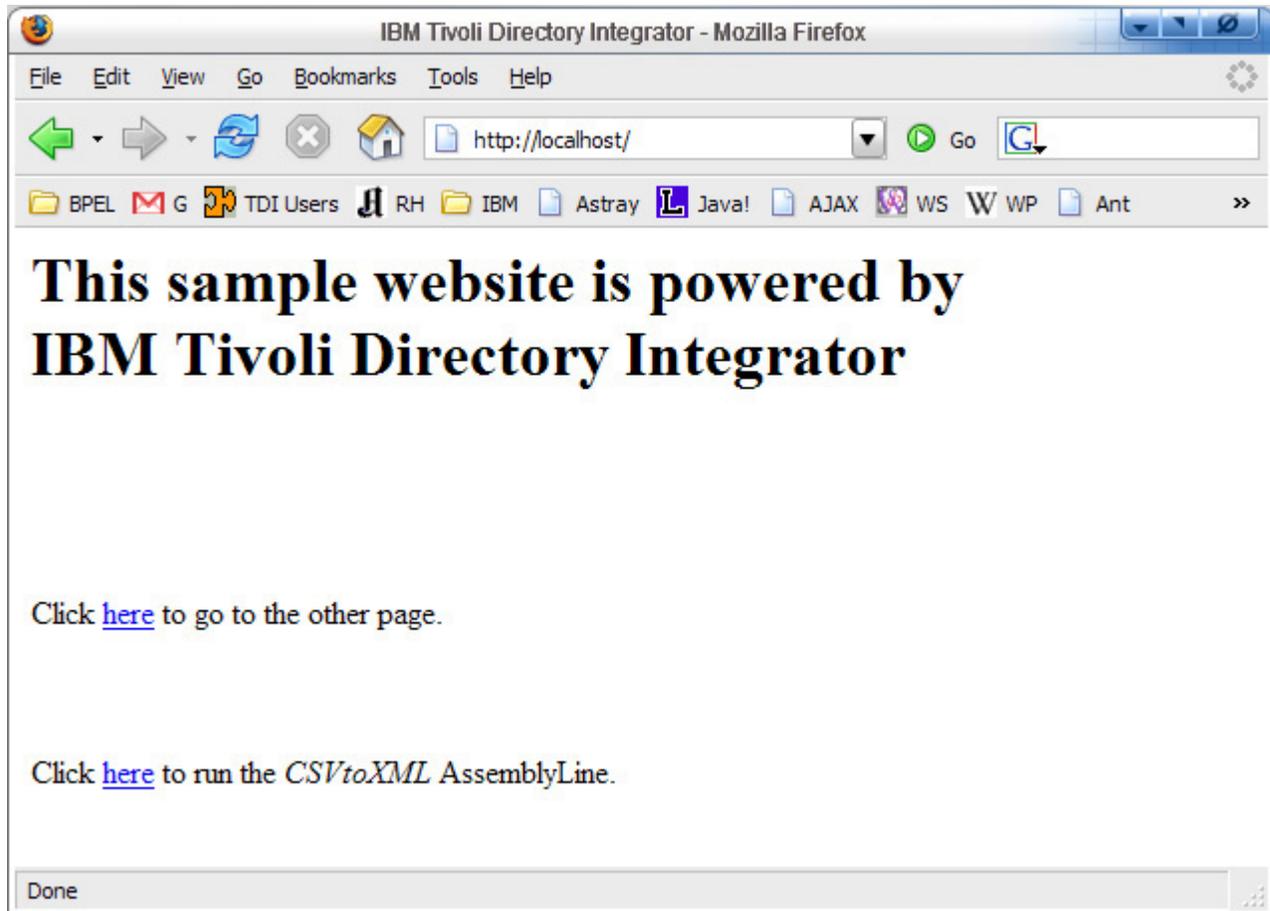
// Finally, you can use the .exists() function to see if the file
// was found. If so, you pass it back in the http.body Attribute.
// If not, you set http.status to "file not found".
//
if ( file.exists() )
    work.setAttribute("http.body", file)
else
    work.setAttribute ("http.status", "NOT FOUND");

```

That was a long snippet, and with a lot of comments. We could of course have implemented this logic as a series of Branches and AttMap components directly in the AssemblyLine. In general, it is best practice to put as much in the AssemblyLine Flow as possible, instead of 'hiding' it away in Hooks. However, sometimes a well-written (and well-named) Script component can keep an AssemblyLine from looking overly complex too.

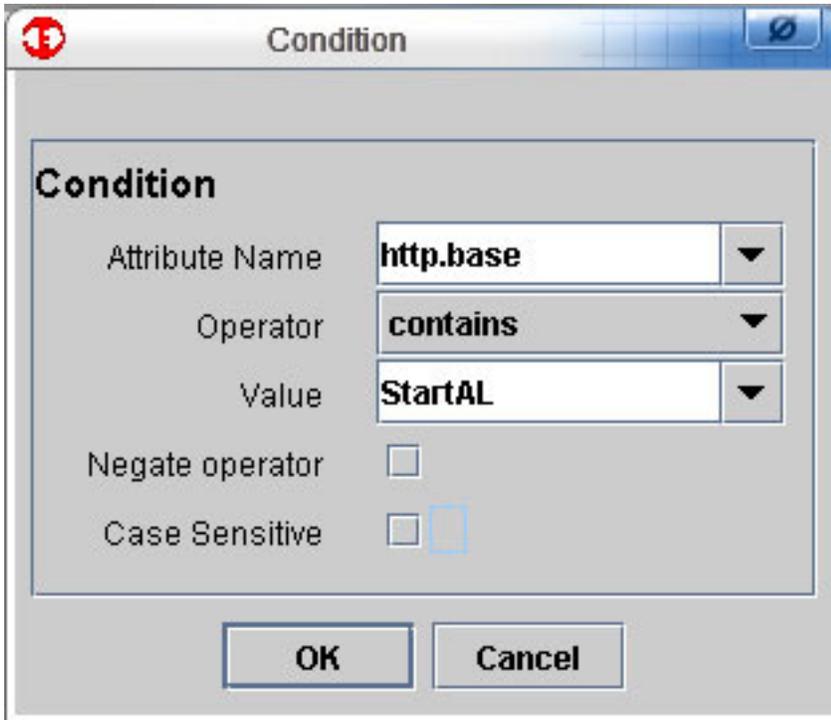
In the Config Editor, click the **Run** button to start your Web server. Dial your Web server again in a Browser. You will see this page:

Screen capture filename: SampleWebsite.eps



In the TDI log you will see that the index.html page was requested. It was a null request, but your scripted code changed this to reference the index page. Clicking on the topmost link will take you to OtherPage.html, which has a link to send you back again.

If you hold the mouse over the link at the bottom of the index page, you should see this text in the browser's status window: `http://localhost/StartAL`. This is a hyperlink to a service that you are about to add to your AssemblyLine. Start by adding a Branch that you call "If StartAL". Drag it above the **ReturnWebPage** SC and give it a simple Condition like this:

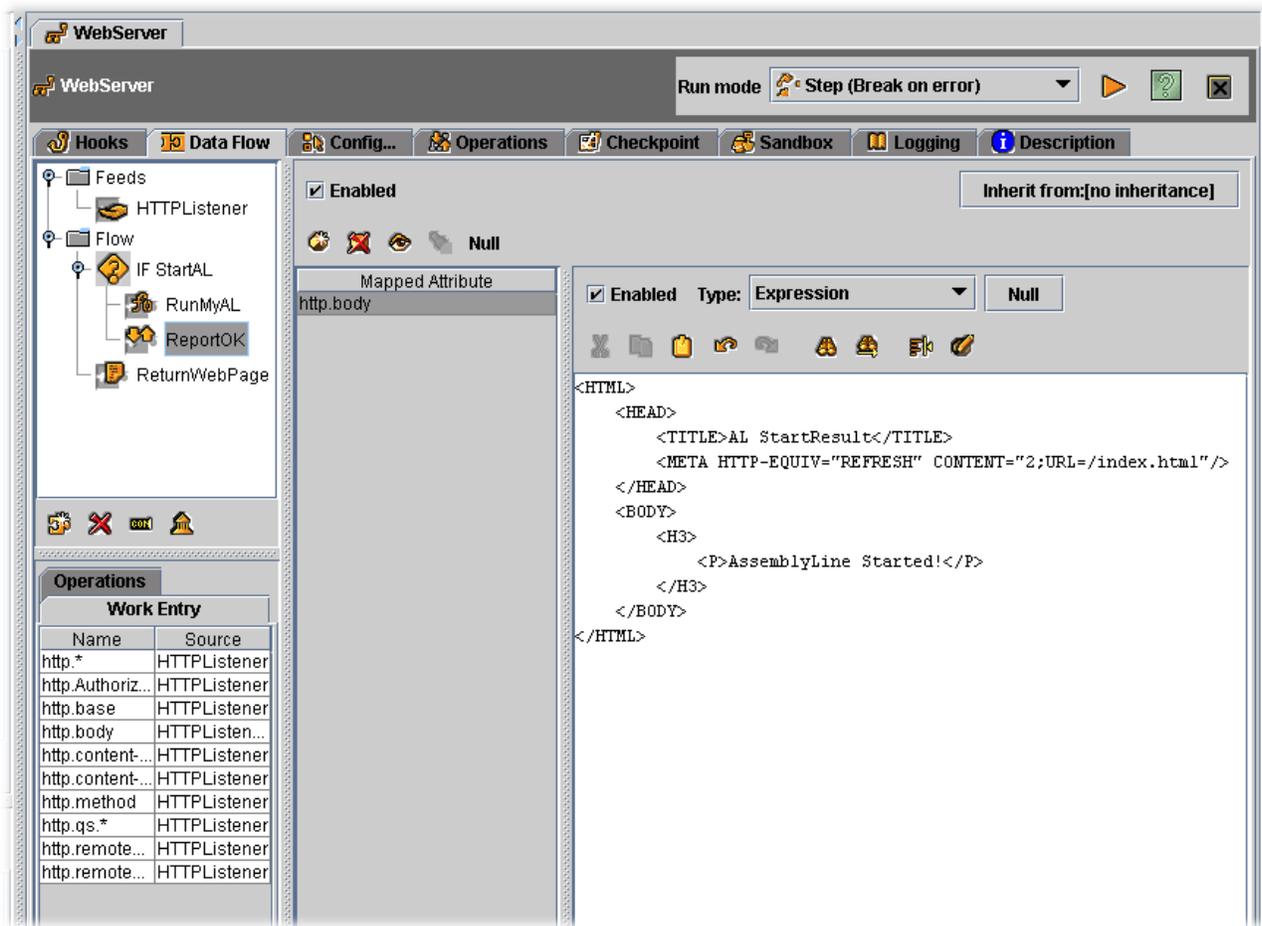


Now right-click on the Branch and add a new Function component. Choose the **AssemblyLine FC** and call it **StartMyAL**.

This Function allows you to launch other AssemblyLines, either on this server, or one on another machine. For this example, simply click on the **AssemblyLine** menu and choose the **CSVtoXML** AssemblyLine. The default setting for Execution Mode is **Run and wait for result**. Change this to **Run in background**, which means that the AssemblyLine you start here will run in parallel with the WebServer AssemblyLine. This lets your main AssemblyLine send a reply back to the client immediately. If you had retained the **Run and wait for result** setting, the browser would hang waiting for a reply until the "CSVtoXML" AssemblyLine completed.

Create this return message by right-clicking on the Branch again and adding an AttMap (attribute map) component called **ReportOK**. Here you add a single Attribute called **http.body**. Set the type of map to **Expression**²⁵. Then you can simply enter the string value (web page) to return:

25. Note that with Expression mapping, you can also embed references to data and configuration settings. See the *IBM Tivoli Directory Integrator 6.1: Users Guide* for more information on this powerful feature.



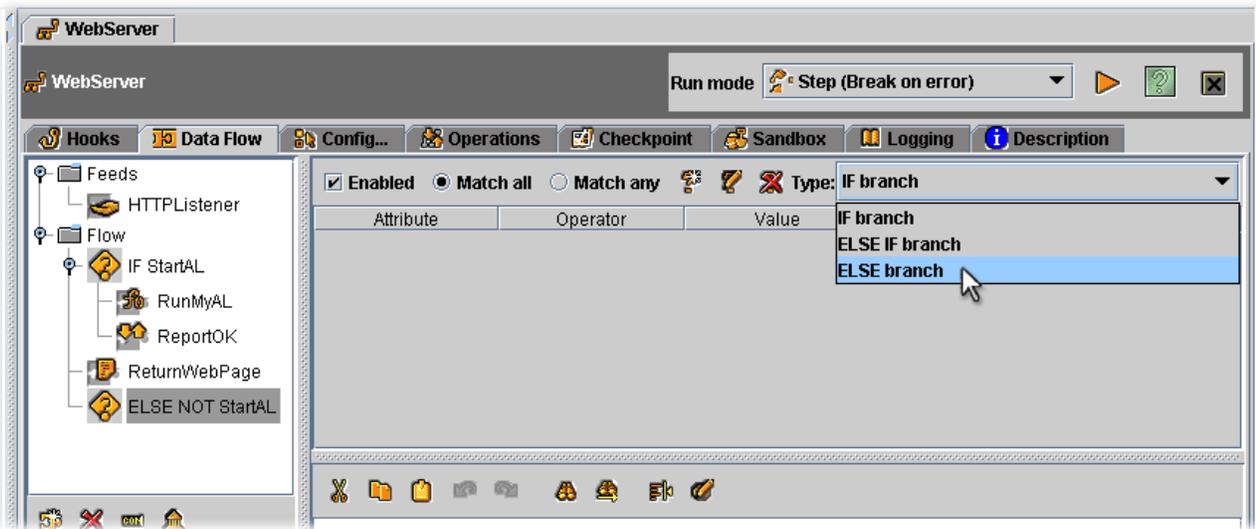
Here is the HTML code for your cutting-and-pasting convenience:

```
<HTML>
<HEAD>
  <TITLE>AL StartResult</TITLE>
  <META HTTP-EQUIV="REFRESH" CONTENT="2;URL=/index.html" />
</HEAD>
<BODY>
  <H3>
    <P>AssemblyLine Started!</P>
  </H3>
</BODY>
</HTML>
```

The <META> tag will cause the page to refresh, going back to index.html.

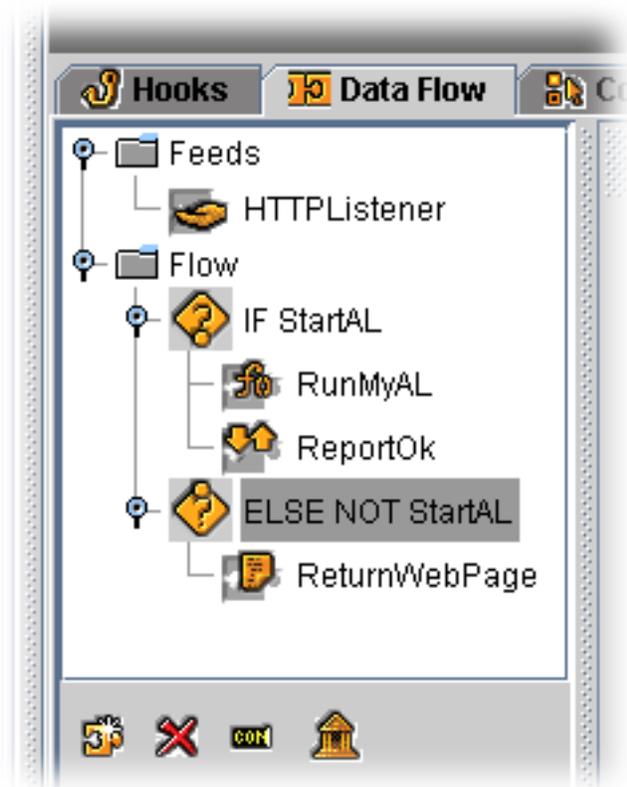
Our solution is almost complete. The only problem we have now is that even if the Branch evaluates to true (i.e. that http.base contains "StartAL"), flow will still end up in the "ReturnWebPage" SC and the contents of http.body will be overwritten. To prevent this, add a second Branch and call it "Else NOT StartAL". Then, in the Details window for the Branch, select the "Else" Type setting.

Screen capture filename: GettingStarted-76.eps



Finally, drag the "ReturnWebPage" Script component on top of this new Branch, which will make it appear under it in the list. Your AssemblyLine components list should look like this now:

Screen capture filename: CompletedAL.eps



Start your AssemblyLine and enjoy your running web server.

Final thoughts

Although IBM Tivoli Directory Integrator makes building data flows fast and easy, the quality of the resulting solution is dependent on your understanding of the problem being solved. Design and implementation decisions are still your responsibility, but IBM Tivoli Directory Integrator can help you over the platform and vendor technology obstacles that otherwise block your vision and limit your imagination.

When you approach an integration problem at the data flow level, you reduce complexity. This gives you gains across the board: in deployment speed, accuracy of the solution, robustness, maintainability, and so forth. In fact, as you start to think in terms of the **simplify and solve** mantra, you will see your infrastructure and its integration possibilities from a whole new perspective.

IBM Tivoli Directory Integrator continues making a difference long after your solutions are completed and deployed. As your business and technical requirements change, IBM Tivoli Directory Integrator lets you adapt and enhance your solutions to meet these new challenges. That's the beauty of IBM Tivoli Directory Integrator: incremental implementation. You can grow your integration solution (and your infrastructure) to fit your needs, as well as the environment where it lives.

Perception is reality, and your perception is formed (and limited) by the toolset you use. The choice is simple. You can continue to accept reality as you perceive it, whittling away at the vision of your integration infrastructure in order to make it fit the tools you are using — or you can switch tools.

Appendix A. index.html and OtherPage.html

You must create two new files in order to complete the examples in this manual: index.html and OtherPage.html.

You must create these files in the examples/Tutorial directory.

index.html

The following is the content of the index.html file.

Copy and paste this code into a flat editor, for example, Notepad, and save the file as **index.html**:

```
<html>
<head>
<title>IBM Tivoli Directory Integrator</title>
</head>
<body>
<h1>This sample website is powered by
<br>IBM Tivoli Directory Integrator</h1>
<br>
<br>
<br>
Click <a href="OtherPage.html">here</a>
to go to the other page.
<br>
<br>
<br>
Click <a href="StartAL">here</a> to run
the <i>CSVtoXML</i> AssemblyLine. </body> </html>
```

OtherPage.html

The following is the content of the OtherPage.html file.

Copy and paste this code into a flat editor, for example, Notepad, and save the file as **OtherPage.html**:

```
<html>
<head>
<title>IBM Tivoli Directory Integrator - Page 2</title>
</head>
<body>
<h1>...and this is the other page.</h1>
</div> <br>
<br>
<br>
<br>
<br>
Click <a href="index.html">here</a>
to go to back to the main page.
</body>
</html>
```

Appendix B. Notices

This information was developed for products and services offered in the U.S.A. IBM might not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the information. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this information at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
Department MU5A46
11301 Burnet Road
Austin, TX 78758
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurement may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

© (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. _enter the year or years_. All rights reserved.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Trademarks

The following terms are trademarks of International Business Machines Corporation in the United States, or other countries, or both:

IBM	Tivoli	AIX®	Lotus
Notes	pSeries®	DB2	WebSphere®
OS/390®	Domino	iNotes	CloudScape

Java, JavaScript and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.

Microsoft and Windows are registered trademarks of Microsoft Corporation.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other company, product, and service names may be trademarks or service marks of others.



Printed in USA

GI11-6480-01

