IBM Initiate Master Data Service
Version 10 Release 0

SDK Reference for Java and Web Services

IBM
Before using this information and the product that it supports, read the information in "Notices and trademarks" on page 139.
# Contents

**Figures** ........................................ v  
**Tables** ................................................ vii  
**Introduction** ......................................... ix  
**Chapter 1. Business and technical overview**  
IBM Initiate Master Data Service review of MDM benefits ........................................ 1  
Industries where MDM can be critical and profitable ..................................................... 1  
MDM scenario at Get Well Hospital .................................................. 2  
Components of the IBM Initiate Master Data Service ........................................ 4  
Master Data Engine ........................................... 5  
MPIINET server ............................................. 5  
Hubs .......................................................... 5  
Clients and applications ............................................. 6  
IBM Initiate Master Data Service and Master Data Engine basics ....................................... 7  
A quick look at Peter Countryman's record .................................................. 7  
Inbound Message processing ............................................. 7  
Standard types of member objects ............................................. 8  
Identifiers used in IBM Initiate Master Data Service software ........................................ 8  
Entity types .................................................. 8  
Relation types ................................................ 9  
Master Data Engine process overview ............................................. 10  
Constructing core member data ............................................. 11  
Derivation process ........................................... 12  
Matching process ............................................ 13  
Comparison process ............................................ 15  
Thresholds ................................................... 15  
Link ........................................................... 17  
Managed task queues .......................................... 17  
Task workflow ................................................ 19  
Dynamic nature of tasks .......................................... 19  
Rules .......................................................... 20  
Versioning – the life cycle of an attribute ............................................. 20  
Using dictionary data ........................................... 21  
User head ....................................................... 21  
Group head ................................................... 21  
Segment head .................................................. 21  
Source head ................................................... 21  
Segment attributes ............................................ 22  
Enumerated data type elements ............................................. 22  
Quiz yourself: architectural and technical overview .................................................. 22  
**Chapter 2. Getting started with the SDK**  
Web services basics ........................................... 25  
Implementing web services ............................................. 26  
Sequence of operations ............................................. 27  
Enabling logging and debugging ............................................. 27  
Web services exposure to the clients ............................................. 27  
IWS data model .............................................. 27  
Installing the SDK ............................................. 28  
Installing the SDK for Java ............................................. 28  
Establishing a connection ............................................ 29  
Making MPIINET connections with Java ............................................. 29  
Disabling Java MPIINET connection KEEPALIVE ............................................. 30  
Workflow of an interaction using the SDK for Java ............................................. 31  
Segments ....................................................... 32  
Representation of MemName segment ............................................. 32  
Dictionary segments ............................................. 33  
Member segments .............................................. 38  
RowList objects .............................................. 40  
RowList structure ............................................. 40  
Example: adding rows to a RowList ............................................. 41  
Example: removing rows from a RowList ............................................. 43  
Example: traversing a RowList ............................................. 43  
Implementation-defined segments ............................................. 44  
Example: creating implementation-defined segments using the SDK for Java ............................................. 44  
Example: creating implementation-defined segments using web services ............................................. 45  
Example: getting data from implementation-defined segments using Java ............................................. 46  
Example: getting data from implementation-defined segments using web services ............................................. 47  
Example: retrieving implementation-defined segments using Java ............................................. 48  
Example: retrieving implementation-defined segments using web services ............................................. 48  
Web services Member class ............................................. 49  
Web services Dictionary class ............................................. 51  
Web services request objects ............................................. 52  
Quiz yourself: getting started ............................................. 52  
SDK for Java overview quiz ............................................. 52  
Web services overview quiz ............................................. 53  
**Chapter 3. Basic interactions** ............................................. 55  
Assumptions about the examples in this book ............................................. 56  
Member interactions ............................................. 56  
Member search .............................................. 57  
Member get ................................................... 60  
Member match ................................................. 62  
Comparing member interactions ............................................. 64  
Member put ................................................... 65  
Qualified member put ............................................. 65  
Member unput ................................................ 65  
Member compare .............................................. 66  
Member merge ................................................ 72  
Member unmerge .............................................. 73  
Member delete ................................................ 74  
Member undelete ............................................. 75  
Member undo .................................................. 75  
Member drop ................................................... 76  
Member score ................................................... 76  
© Copyright IBM Corp. 1995, 2011
Figures

1. Components of IBM Initiate Master Data Service .................. 5
2. Inbound Broker processing ........................................ 8
3. Basic Master Data Engine processes .......................... 11
4. Derivation process .............................................. 13
5. Matching process .............................................. 14
6. Scoring thresholds .............................................. 16
7. API interaction ................................................. 25
8. Web services Architecture ....................................... 26
9. Interaction using MemRowList .................................. 31
10. Representation of MemName segment .......................... 32
11. Structure of MemRowList ....................................... 41
<table>
<thead>
<tr>
<th>No.</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Base IBM Initiate Master Data Service</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>Components and Applications</td>
<td></td>
</tr>
<tr>
<td>2.</td>
<td>Sample Patient Record</td>
<td>7</td>
</tr>
<tr>
<td>3.</td>
<td>Sample Patient Record</td>
<td>12</td>
</tr>
<tr>
<td>4.</td>
<td>Potential candidate records</td>
<td>15</td>
</tr>
<tr>
<td>5.</td>
<td>Example of Potential Overlay</td>
<td>18</td>
</tr>
<tr>
<td>6.</td>
<td>Example of Potential Duplicate</td>
<td>18</td>
</tr>
<tr>
<td>7.</td>
<td>Example of Potential Linkage</td>
<td>18</td>
</tr>
<tr>
<td>8.</td>
<td>Tasks</td>
<td>20</td>
</tr>
<tr>
<td>9.</td>
<td>MemName Segment</td>
<td>32</td>
</tr>
<tr>
<td>10.</td>
<td>Type Definitions</td>
<td>33</td>
</tr>
<tr>
<td>11.</td>
<td>Segment Definitions</td>
<td>34</td>
</tr>
<tr>
<td>12.</td>
<td>Source Definitions</td>
<td>35</td>
</tr>
<tr>
<td>13.</td>
<td>Use Definitions</td>
<td>36</td>
</tr>
<tr>
<td>14.</td>
<td>User Access Definitions</td>
<td>37</td>
</tr>
<tr>
<td>15.</td>
<td>Relationship Segment Definitions</td>
<td>38</td>
</tr>
<tr>
<td>16.</td>
<td>Member Segments</td>
<td>38</td>
</tr>
<tr>
<td>17.</td>
<td>Comparing Member Interactions</td>
<td>64</td>
</tr>
<tr>
<td>18.</td>
<td>memCompReq Tag</td>
<td>69</td>
</tr>
<tr>
<td>19.</td>
<td>memHead Tag</td>
<td>70</td>
</tr>
<tr>
<td>20.</td>
<td>Example Customer Member Attributes</td>
<td>129</td>
</tr>
<tr>
<td>21.</td>
<td>MemPut Options</td>
<td>132</td>
</tr>
<tr>
<td>22.</td>
<td>MemPut Match Mode Options</td>
<td>133</td>
</tr>
<tr>
<td>23.</td>
<td>Configuration Settings</td>
<td>133</td>
</tr>
<tr>
<td>24.</td>
<td>Error Codes</td>
<td>135</td>
</tr>
<tr>
<td>25.</td>
<td>IBM resources</td>
<td>147</td>
</tr>
<tr>
<td>26.</td>
<td>Providing feedback to IBM</td>
<td>147</td>
</tr>
</tbody>
</table>
Introduction

This reference includes detailed information about using the software development kits for Java and Web Services to create custom solutions and applications for the Initiate® Master Data Engine.

The reference covers the following topics:

- Basics of Master Data Management (MDM)
- Introduction to IBM® Initiate Master Data Service and the Master Data Engine
- Complete overview of the SDK offerings for Java and web services

System requirements

Your development environment must meet the following requirements:

- Master Data Engine is installed
- The applicable SDK is installed
- Hardware and software as specified in the IBM Initiate Master Data Service
  System Requirements document

For additional installation and configuration requirements for each SDK, refer to the applicable section of this guide.

Note: Any applications written and compiled with a pre-7.5 version of IBM Initiate SDK must be recompiled with a 7.5 or later SDK for compatibility with the core Master Data Engine and 64-bit identifiers.
Master Data Management (MDM) is the dynamic and intelligent linking of an organization's data. The data to be linked can reside in disparate applications, databases, and locations inside and outside of the organization. By using MDM software and intelligently linking this customer data, an organization can provide a more complete and intelligent view of this information. Furthermore, this linked data can be used to provide a 360-degree view of a customer, to support personalized customer interactions, and to uncover trends and patterns that provide insight into gaining increased profitability.

IBM Initiate Master Data Service review of MDM benefits

IBM Initiate Master Data Service accurately identifies and links the records of customers to drive an on-demand enterprise. IBM Initiate Master Data Service components can help you turn any volume of data into the clean, complete personal profiles you need to drive better, more cost-effective and robust interactions with customers, patients, partners and organizations.

IBM Initiate Master Data Service and the core Master Data Engine use advanced statistical and database techniques to accurately score, match and link customer data across your enterprise.

In summary, the IBM Initiate Master Data Service accomplishes the following:

- Breakdown of barriers between systems
- Linking of data on demand in real-time, near real-time, or batch
- Delivery of the most accurate results
- Scalability to work with any volume of structured data from any source
- Matching of implementation to your needs

In addition, IBM Initiate Master Data Service does not force standardization of data or require changes to customer identifiers in any database or system.

Industries where MDM can be critical and profitable

The list of industries where the implementation of a MDM solution can influence both profit and customer service and satisfaction is almost limitless. Some of these areas include the following:

- Health care
- Financial services, banking, and insurance
- Pharmaceutical, including retail pharmacies
- Hospitality
- Public sector
- Transportation and logistics
- Manufacturing
- Banking

Note: All proper names used in the example scenarios in this reference are fictitious. Any similarity to the real names of persons or business entities is coincidental and unintentional.
MDM scenario at Get Well Hospital

The following hypothetical scenario in fictitious “Get Well Hospital” helps illustrates the role of MDM in organizations.

Get Well Hospital is located in the heart of a large metropolitan area. A large corporate hospital, it offers several specialty services for patients throughout a six-county area. The hospital has implemented its electronic patient information database in a phased manner, one department at a time.

Over a span of ten years a patient named Peter Countryman was treated in various departments of Get Well Hospital. Each department maintained a separate database with relevant details of Peter Countryman and other patients similar to him. During the course of his treatments, Peter Countryman changed his residence several times. Each time he used the services of a different department, he gave his latest address and telephone number. Therefore, some departments stored obsolete address information while others recorded newer details. Over the years several other patients with name combinations of “Peter” and “Countryman” received the services of these medical departments. Some of these patients also changed residences, and always provided their latest phone numbers, addresses, and other information with each visit.

After several years of treatment, a team of surgeons at Get Well Hospital requires the complete medical history of Peter Countryman from across all departments to evaluate his fitness for surgery.

Typically when a patient registers himself in a department, he is asked to provide some personally identifiable information. Most departments take only the relevant patient information and store it in a unique structure. For instance, each department might store a patient’s “name,” but in a format different from other departments. Consider the case of Peter Countryman. The Dental department stored this patient’s name as P Countryman, while the Endocrinology department stored his name as Peter Countryman. The surgery team, in trying to get a complete medical history of Peter Countryman from across all departments, is faced with the first challenge of identifying the correct medical records for Peter Countryman from among the numerous and similar medical records in each department.

For example, how would you differentiate between two medical records in the Dental department that have the same patient name as P Countryman? How can you be sure a particular medical record belongs to this Peter Countryman and not to another patient called Pete Countryman? The solution is to compare Peter Countryman’s other identifiable attributes, such as social security number, address, and telephone number, with those of Pete Countryman.

Background information

Patient information is fragmented. In our scenario, each department had partial information for Peter Countryman. Besides his name, some departments had stored only his social security number and address, while others stored his telephone number and postal code. None of the departments had a complete set of details regarding Peter Countryman.

Duplicate records exist. Duplicate records were created when Peter Countryman was asked to register again on his second visit to a particular department at the Regional Medical Center of Get Well Hospital because he had forgotten his registration number.
As mentioned above, these individual departments typically have their own patient databases. In many cases they were originally independent business entities, but later business dealings made them part of the same corporation. This association can present the following challenges:

- Databases are not linked with each other.
- The database software across these departments might be different.
- Database structures are probably different.

To compound the problem, there are several other patients sharing the name of “Peter” or “Countryman” within the multiple databases. Some of these members have details and attributes similar to those of Peter Countryman.

**Software required to resolve the situation**

Any proposed software solution should not only break the barriers between the disparate systems of the departments, but also compare Peter Countryman’s personally identifiable attributes with the attributes of other members and accurately identify Peter Countryman’s medical records in each of the multiple departments. It should then link these respective records in the multiple departments and databases so that a complete profile of Peter Countryman is available to the surgeons, and to all departments in Get Well Hospital.

**Note:** In MDM for health care, we use only the personally identifiable attributes of members to identify and link member records. We typically do not use other information, such as suggested medication and X-ray results, to compare members.

**IBM Initiate Master Data Service in action**

IBM Initiate Master Data Service is deployed at Get Well Hospital to resolve this situation.

The Hub, and ultimately the Master Data Engine, receive data in either real-time or batch from individual systems by way of relational database connections, APIs, web services, messaging solutions, or flat files. These source systems provide the data that is used for scoring, matching, and linking members within the IBM Initiate Master Data Service software.

The Master Data Engine runs the source data through a derivation routine. Derivation is the process of extracting demographic data elements to be used in scoring and matching. The derived data is then stored in a highly optimized format in a relational database.

- The cornerstone of the IBM Initiate Master Data Service platform is the set of intelligent algorithms provided by your selected Hub configuration. These algorithms use a series of statistical formulas and customizable parameters to rapidly evaluate the data and provide member matches. These matches can be used to form linkages that form member entities.
- IBM Initiate Master Data Service software marks exceptions and data entities that fall outside the defined matching parameters as possible matches, and creates tasks for later review and resolution. This process, known as data remediation, is fully supported in the solution.

**Benefits to Get Well Hospital**

By implementing the IBM Initiate Master Data Service components, Get Well Hospital received the following benefits.

- The Master Data Engine and associated algorithms accurately identify Peter Countryman’s medical records from records of other patients whose records have similar details.
Surgeons and physicians get a clear picture of the patient's medical history.
A 360-degree view of each patient is now available on demand to all
departments in the hospital.
Clear insights on trends are now available on demand.
If Peter Countryman changes his address or other personal information at one
department or facility, all other departments can receive the update.

Components of the IBM Initiate Master Data Service

IBM Initiate Master Data Service and the core Master Data Engine compose the
critical link between data sources and an operational application. The Master Data
Engine receives data in real-time or from batch files from individual records,
internal data sources, business partners or other third-party sources. Data comes
into the Master Data Engine as it appears on its native systems (sources). The
Master Data Engine processes then derive new data that enables the Master Data
Engine to perform fast and accurate linking.

Table 1 and Figure 1 on page 5 present textual and graphical representations of the
architecture.

Table 1. Base IBM Initiate Master Data Service Components and Applications

<table>
<thead>
<tr>
<th>Component/Application</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Master Data Engine</td>
<td>Houses the logic, algorithms, and processes to search, compare, score, and link members and manage relationships. Has a relational database management system (RDBMS) as a back end. Within the Master Data Engine lives your Hub environment, for example, IBM Initiate Patient, IBM Initiate Provider, or Initiate Consumer.</td>
</tr>
<tr>
<td>IBM Initiate Workbench</td>
<td>An Eclipse-based client employed for Hub security, dictionary, and algorithm configuration, as well as data analytics tools.</td>
</tr>
<tr>
<td>Enterprise Viewer</td>
<td>Web-based client application that enables searching and retrieving of member information.</td>
</tr>
<tr>
<td>IBM Initiate Inspector</td>
<td>Web-based client that enables task resolution, data management, and relationship management.</td>
</tr>
<tr>
<td>Reports</td>
<td>Web-based operational reports about IBM Initiate Master Data Service software activity.</td>
</tr>
</tbody>
</table>

Though not part of the standard IBM Initiate Master Data Service offering, the
Message Broker Suite integration component is typically used to pass data from
sources to the IBM Initiate Master Data Service and back again.

It is important to understand the composition of IBM Initiate Master Data Service
and what some of the internal entities, objects, and processes are, as illustrated in
Figure 1 on page 5.
On the left are the integration components: the Java, Microsoft .NET, and web services Application Programmer Interfaces (APIs).

In the center is the Master Data Engine, used for processing and intelligently linking all the identified data.

At the right is the relational data store, which manages IBM Initiate Master Data Service related data and records.

Master Data Engine

The core Master Data Engine contains the logic that is at the heart of the IBM Initiate Master Data Service product. The logic, rules, and algorithms configured for each Hub implementation enable the Master Data Engine to compare member records and produce scores that indicate which records are likely to represent the same entity or which individual members form relationships with other individual members, and the relative strength of the comparison.

MPINET server

The MPINET server provides TCP/IP socket connections for the various API calls from the clients. The server provides an optimized proprietary communication protocol internally used by the IBM Initiate SDKs and the core Master Data Engine. The server is designed to handle multiple socket connections, is multithreaded, and implements database connection pooling for optimum performance and fast response times.

Hubs

Within each Master Data Engine installation, you can configure multiple Hubs. Encompassed within your Hub are:

- Data attributes that are specific to your domain
- Customized algorithms that are processed by the Master Data Engine
- Rule definitions that the Master Data Engine uses to form entities and relationships between records

The flexibility of IBM Initiate Master Data Service architecture enables organizations to configure Hubs that meet their unique data and business needs.
Hubs enable enterprises to fully understand the relationships between members and records across data domains and within the same domain. Some examples of Hubs follow.

- Patient at Provider
- Consumer at Provider
- Contact at Organization
- Consumer at Organization
- Hierarchy
- Household

Peter Countryman is a customer of a large cellular phone company, Cell&Cell. Peter uses his Cell&Cell phone for business only and has arranged for his employer, ReallyBigCorp, to pay his cellular phone bill directly. Cell&Cell has implemented the Consumer at Organization Hub to determine how much business ReallyBigCorp represents. Typically, because Peter Countryman is the customer, Cell&Cell would have a hard time determining how much business it does with ReallyBigCorp. By implementing a Hub, Cell&Cell can determine exactly how much business ReallyBigCorp represents.

**Clients and applications**

The Initiate client applications that are available include the following.

**IBM Initiate Workbench**

IBM Initiate Workbench, delivered with the base IBM Initiate Master Data Service package, provides capabilities for user management and configuration of the software, specifically the data dictionary (data model) and algorithms that define a Hub. From IBM Initiate Workbench you can define member and entity types, sources, applications, attributes, task threshold levels and algorithms.

**Enterprise Viewer**

This Web-based client application enables the searching and retrieving of member information. Surgeons at Get Well Hospital can use Enterprise Viewer to search for records for Peter Countryman and other patients by submitting the desired criteria via a Web client. Users are able to attach notes to member records, but otherwise, Enterprise Viewer is a read-only application; no information contained in the Master Data Engine Hub can be altered through the application.

**IBM Initiate Inspector**

This is a web-based client that enables users to work tasks created by users or by the Master Data Engine processes. Member records are automatically linked when their score is above the Autolink threshold. Members that compare between the Clerical Review and Autolink thresholds produce Potential Duplicate or Potential Linkage tasks, indicating that enough information is available to determine the members are similar, but require a data analyst (data steward) to make the final decision. Other task types include Potential Overlay and Review Identifier tasks. Additionally, graphical relationships and hierarchies between members and organizations enable users to better understand their data.

Read more about tasks in “Managed task queues” on page 17.
Reports

Web-based operational reports enable a quick method for reviewing Master Data Engine and Hub events. For example, you can view details about events that are triggering task creation, the number of linkages occurring, the number and type of tasks being resolved, tasks that remain unresolved, and the users who are working tasks.

IBM Initiate Master Data Service and Master Data Engine basics

The remainder of this section provides some basic concepts that will help you understand how Initiate software works.

A quick look at Peter Countryman's record

Let's take a quick look at the record created when Peter Countryman registers at the Get Well Hospital orthopedic department.

Table 2. Sample Patient Record

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>First name</td>
<td>Peter</td>
</tr>
<tr>
<td>Last name</td>
<td>Countryman</td>
</tr>
<tr>
<td>Gender</td>
<td>Male</td>
</tr>
<tr>
<td>Address</td>
<td>1N Oak Street</td>
</tr>
<tr>
<td>City</td>
<td>Phoenix</td>
</tr>
<tr>
<td>Zip code</td>
<td>85001</td>
</tr>
<tr>
<td>Phone number</td>
<td>602 911 9111</td>
</tr>
<tr>
<td>E-mail address</td>
<td><a href="mailto:Peter@xyz.com">Peter@xyz.com</a></td>
</tr>
<tr>
<td>SSN</td>
<td>007007007</td>
</tr>
<tr>
<td>Source ID</td>
<td>AK</td>
</tr>
<tr>
<td>Member ID</td>
<td>2345</td>
</tr>
</tbody>
</table>

We will use this example to see how IBM Initiate Master Data Service dynamically and intelligently matches the record we just obtained with the information we already know about Peter Countryman.

Inbound Message processing

Typically after a source system has been identified and is available, information will pass from the source system to the Inbound Broker. As shown in Figure 2 on page 8, the source system (which in our scenario is the orthopedic department's system), passes the Peter Countryman record to the Inbound Broker. The Inbound Broker receives the data and conceptually has three parts: a reader, a process engine, and access to the Master Data Engine. The main function of the Inbound Broker Service is to efficiently feed the incoming data to the Master Data Engine via MPINET.
The Inbound Broker processing diagram illustrates how data flows from the datasource through the inbound broker, which consists of the reader, process engine and API, and then to the Master Data Engine via MPINET.

The Inbound Broker receives the incoming data via the Inbound Reader, which enables the algorithms within the Master Data Engine to perform fast and efficient linking. The Inbound Reader, whose main function is to receive the data and acknowledge its receipt, then works in conjunction with the Inbound Broker process to parse the personal attributes, such as name and phone number, contained in the incoming data. The attributes gathered from the incoming data are then used within the Master Data Engine to create the definition of a member object. Member objects, their roles, and the various types are all discussed in the next section.

**Standard types of member objects**

A member object is the basic unit in IBM Initiate Master Data Service—the lowest represented information whose identity is to be established. Member types can be directly mapped in concept with a Hub type. Within IBM Initiate Master Data Service software, we provide Hub configuration templates, which define common types of member objects (Person, Patient, Provider, Consumer, Citizen, Banking, and Organization) out of the box. Additionally, IBM Initiate Master Data Service software enables organizations to configure any other type of member object to properly represent their data. For example, some law enforcement customers have the need to compare an individual’s height, weight, eye color, and hair color to create a more accurate list of suspects.

**Identifiers used in IBM Initiate Master Data Service software**

It is equally important to understand how IBM Initiate Master Data Service software uniquely identifies each member object. Without properly and uniquely identifying each object, selecting, comparing, scoring, and linking become a very difficult task. IBM Initiate Master Data Service software uses both unique keys provided by each source system and a set of unique keys—based upon the Master Data Engine processing—to identify participating objects. Below we describe four of the common keys used to identify member objects within IBM Initiate Master Data Service.

Developers will need to pay special attention to use of “int” and “long” when dealing with key attributes (audrecno, entrecno, memrecno, setrecno, bkthash, bktval, bktrole, and seqnum). Because these are 64-bit integers, you will want to make sure that you do not down cast the “long” into an “int” type.
Member ID (MemIdnum)

Source systems, like the orthopedic department’s system in our example, send data to the IBM Initiate Master Data Service for processing. In general, these source systems uniquely identify each of their own records for easy processing and retrieval. The Master Data Engine captures these unique record identifiers from each source system and associates it as that record’s member ID within the Hub. Member IDs are then used within IBM Initiate Master Data Service components for processing, retrieval, and updates. Note that single and double quotes are not valid characters in a memIdnum.

Source ID (SrcCode)

Having a unique member ID helps IBM Initiate Master Data Service understand individual records, but also track where that record originated, further enhancing the usability of the data. Source IDs are unique IDs that identify each data source individually. The unique combination of member ID and source ID enables Initiate solutions to see from which system a record originated. This ID can be used for synchronization and changes to that record.

Enterprise ID (EID or EntRecno)

The Master Data Engine assigns Enterprise IDs (EID) to member objects. Enterprise IDs are generated as a result of the Master Data Engine's scoring and linking process. All members sharing an Enterprise ID represent an entity relationship (refer to "Entity types") for that record.

As an example, if there are two member records for Peter Countryman having the same Enterprise ID with an entity type of Identity, then the Master Data Engine has determined that these two records represent the same member object.

Member Record Number (MemRecno)

The Master Data Engine processes assigns this number to the member's record. This value is unique for each record.

Note: Avoid storing the Enterprise ID and Member Record Number in other software systems. Since the IDs are not static, when members change, the Enterprise ID can also change. Any member can always be known by its SourceID/MemIdnum, and any linkage set can be known by any one of the set's members.

For example, when a new member's record is added to the system, the Enterprise ID for the member will be the same as the MemRecno. If another record is added for the same member in a different source, these two records will be linked and both records will be assigned the same Enterprise ID (indicating that there is a linkage).

Entity types

An entity is defined as “something that exists as a particular and discrete unit.” In terms of identity and data management, an entity is the logical relationship between two or more records. Entities are represented in the IBM Initiate Master Data Service software as records sharing a common Enterprise ID. An entity can also be called a linkage set. There can be an unlimited number of records in an entity or linkage set.
An entity type allows for distinction between the way members are viewed and linked within IBM Initiate Master Data Service. Each entity type has a specific algorithm configuration. Examples of entity types are described below.

**Identity**

The first type of entity is Identity. Records from the various source systems that belong to the same member are identified by the matching algorithm and linked with a common Enterprise ID. When an organization wishes to link member records of individual members like Peter Countryman's, then it can use the Identity entity. In this case, the algorithms compare the records to determine if the record belongs to an individual. For instance, all records of Peter Countryman are identified in all of the departments and linked; all records of Roberta Salmon are identified and linked, and so on.

**Household**

Another type of entity is Household, in which all the members' information is linked together based on a location-oriented algorithm. For example, consider Peter Countryman's family: Mr. Peter Countryman; his wife, Mrs. Mary Countryman; and their son, Frank Countryman.

Peter Countryman has a record in the ophthalmology department of Get Well Hospital, and Mrs. Mary Countryman and Frank Countryman have their records in the dental department. They would all be linked as one entity known as Household. In this case the algorithms would not only check to verify that the record belongs to one individual for the Identity entity, but would also check to see whether the record belongs within the same household as another member. This would be useful, for instance, when records of all the family members might be required by a hospital staff to check hereditary diseases.

**Group**

In normal processing, a record can only belong to a single entity within an entity type. Group entity functionality enables records to have multiple entity record numbers (entRecno) within a single entity type. A member of a group entity must match all members of that entity set above the auto-link threshold.

**Relationship types**

There are two types of relationships: entity-to-entity and group.

**Entity-to-entity**

The relationship between entities and members, and such relationships are considered transitive.

**Group relationships**

Individual members grouped by a common attribute, such as address or employer, into a non-transitive entity.

---

**Master Data Engine process overview**

Returning to our scenario at Get Well Hospital, the record for Peter Countryman from the incoming orthopedic department has been received by the Inbound Message-Based Transaction Service and is now being passed to the Master Data Engine for processing.
As shown in Figure 3, the Master Data Engine is logically divided into three processes:
1. construct core member data
2. derivation process
3. matching process

The Inbound Broker passes data to the Master Data Engine. The Master Data Engine then constructs and stores the member object. Following this, the Master Data Engine:
• Standardizes the data from the source record
• Intelligently derives new data from the incoming record
• Selects other records already in Master Data Engine database that might have the same information
• Compares the information attribute-by-attribute from each record; note that only attributes defined as “comparable” in the algorithms are used
• Generates a score for the incoming records
• Dynamically links records that meet the business threshold

The configurable algorithms are used by the Master Data Engine to create the derivation data, compare member records and produce scores that indicate which records are likely to represent the same entity and the relative strength of the comparison.

**Constructing core member data**

As our record from the orthopedic department's system (see table below) enters the Master Data Engine, the Engine first constructs the core member object used to identify and compare the records of the system. Constructing core member object involves storing the parsed data from the Inbound Broker.

**Verticalization**

The construction of core member data is actually a unique “verticalization” process, where all relevant attributes are stored as separate rows, and possibly in separate tables, in the Master Data Engine database. Consider our example Table 3 on page 12 of the record created for Peter Countryman in the orthopedic system:
Table 3. Sample Patient Record

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>First name</td>
<td>Peter</td>
</tr>
<tr>
<td>Last name</td>
<td>Countryman</td>
</tr>
<tr>
<td>Gender</td>
<td>Male</td>
</tr>
<tr>
<td>Address</td>
<td>1N Oak Street</td>
</tr>
<tr>
<td>City</td>
<td>Phoenix</td>
</tr>
<tr>
<td>Zip code</td>
<td>85001</td>
</tr>
<tr>
<td>Phone number</td>
<td>602 911 9111</td>
</tr>
<tr>
<td>E-mail address</td>
<td><a href="mailto:Peter@xyz.com">Peter@xyz.com</a></td>
</tr>
<tr>
<td>SSN</td>
<td>007007007</td>
</tr>
<tr>
<td>Source ID</td>
<td>AK</td>
</tr>
<tr>
<td>Member ID</td>
<td>2345</td>
</tr>
</tbody>
</table>

Remember that from the above information, the Inbound Broker has created unique packets of information to be handed to the “construct core member” process. The construct core member process writes the data to the associated tables in the database.

**Benefits of verticalization**

The Master Data Engine architecture employs several unique child tables, each to store individual attributes of Peter Countryman such as name, gender, and phone, for improved computational speed, additional processing, and express access.

Attributes are stored in the various database tables, for example, the member address table, member name table, and member phone table. Thus, unlike a typical data structure where each member has one row of data, the information is stored in separate tables. Each member object can have more than one row in each of these tables.

This is the foundation for IBM Initiate Master Data Service software that makes the data model so robust. In today’s culture, in which individuals have multiple phone numbers and e-mail addresses, such a model makes it easy to store the data.

IBM Initiate Master Data Service stores and manages data within industry standard databases. The schema is designed to be flexible and efficient when selecting, comparing, scoring, and linking the vast amounts of data across an enterprise. The Initiate software also enables customers to exploit many of the high availability and scalability features natively provided by each of these vendors, such as Oracle Real Application Clusters or IBM High Availability Clusters Multi-Processing.

**Derivation process**

After the core member data is created and written, the next step within the Master Data Engine is the derivation process as shown in Figure 4 on page 13. The derivation process within the Master Data Engine can actually be broken up into three logical sub-processes:

1. standardizing of the incoming data
2. bucketing data, or segregating new data entities into specialized buckets
3. comparing data, or the extraction of data from the incoming source for comparison purposes

**Standardization**
Standardization is the resolution of certain attributes within the data to a common format. For example, the name “O’Malley” could be standardized to “OMALLEY” after removing the punctuation and changing all characters to uppercase.

**Bucketing**
After the standardization process is complete, the derivation process performs a bucketing process on the data. In this process, attributes, which will form the various buckets that are identified during the initial configuration of Master Data Engine and Hub, are grouped together. Typically, a solution requires about five to seven buckets per member. In each bucket, there could be one or more attributes involved. For example, the buckets might be [first name], [last name and zip code] and [street name]. Buckets help in narrowing the search by identifying a group of candidate records who share some of the same bucketing information. Using our example of Peter Countryman with the record from the orthopedic department, we might have bucket information as, Peter, Countryman85001 and 1N Oak Street.

**Comparing**
The final step in the derivation process is the generation of additional comparison data. All the attributes that are used in the comparison process are extracted from the core data and stored separately in the derived data layer. Now we have two types of data, which constitute the derived data layer. First is the bucket data, used in the candidate selection process, and second is the candidate comparison data, used when comparing two specific members.

**Matching process**
After Master Data Engine has completed the derivation process, we can begin to apply meaningful matching and comparison algorithms to the data to determine whether the incoming record is or is not related to other records already processed in Master Data Engine database (refer to Figure 5 on page 14). The algorithms take
full advantage of the derived bucketed and derived comparison data created in the derivation process above.

The key to the Master Data Engine is in how it compares and achieves a finer definition of how two or more records might be related. Other solutions stop at the highest level of comparison and can lead to what is known as many “false positives” because of the way they store the data and the limited matching capability. The Master Data Engine software is different because its matching process involves logical steps:

1. Selecting a group of candidate records.
2. Comparing each of the candidate records to the incoming record attribute by attribute.
3. Scoring the incoming record.
4. Linking records that meet the established business criteria.

When the score is above the autolink threshold, the records are automatically linked. When the score is between the clerical review threshold and the autolink threshold, they are placed into task queues for human review. When the score is below the clerical review threshold, the Master Data Engine concludes that the records do not belong to the same member, and no task is created.

**Select**

This portion of the process involves the selections of candidate records from the records already received and scored within the Master Data Engine. Candidate selection is the collection of known records that will be further analyzed in the comparison process. As stated earlier, the Master Data Engine calculated a series of values useful for comparison for our incoming record—“Peter,” “Countryman85001” and “1NOakStreet”—and stored them in the database. The process uses each of these values to compare against the values of the already existing member’s objects in the database and creates a list of possible candidates who share the bucketing information. Table 4 on page 15 is an example of the list of potential candidate records that might be passed to the comparison sub process.
### Comparison process

The two logical steps in the comparison process are compare and score. These two steps differentiate IBM Initiate Master Data Service from all other approaches.

#### Compare

This step actually compares the individual attributes gathered at derivation time, attribute-by-attribute, against the records already in a Hub. The Hub contains a library of comparison functions that have been developed over years of experience in matching data. These functions determine the degree of similarity between the attribute values. The results from these comparisons are passed to the scoring process.

In the first example above, the comparison would show that the last names did not match, the first names did match on Peter, and the addresses were different.

#### Score

The results of the comparison are given a numerical score to indicate the likelihood that the records refer to the same member. Scores can be negative (implying non-relationship) or positive (implying relationship). The greater the score, the more similar the members are to each other.

The Master Data Engine utilizes the score of the candidates to indicate the next processing step. When automatic linkage is turned on during insert and update operations, three defined thresholds determine the actions the Master Data Engine will take based on the resulting score when comparing members. The threshold for each solution can be adjusted based on the business case that needs to be solved.

### Thresholds

*Thresholds* are scoring levels set to determine how records will be managed by the Master Data Engine. There are three thresholds.

#### Auto-link threshold

As shown in Figure 6 on page 16 by the squares to the right of the auto-link (AL) threshold line, when the value of the score of two records is above the auto-link threshold, the Master Data Engine automatically links these records. This value indicates that the particular member object already exists in the database and therefore the member is automatically grouped/linked with other member records.
and assigned a common Enterprise ID. Note that the data is not merged, but is only linked.

![Example scoring thresholds](image)

**Figure 6. Scoring thresholds**

In the scoring thresholds figure, the triangles represent records that should not be linked because they represent unique members. The squares represent records that should be linked because they represent members already in the system. Only the records that should be linked fall above the autolink threshold. Some records that should be linked and some records that should not be linked will fall in the manual review band. Most records that should not be linked fall below the task or clerical review threshold.

**Task threshold (Clerical Review)**

The values in the Manual Review band are below the auto-link threshold but above the business acceptable lowest boundary or task threshold. When the calculated score is between the task threshold and auto-link threshold, the record is thought to be a high enough degree of similarity to warrant manual review. The resulting action is to trigger an “issue or task record” for review. Through the use of Inspector for data resolution, a user can determine whether these records are duplicate records and whether they should be linked.

**Overlay threshold**

The value of overlay threshold (OVL) is configurable and tends to be a negative value. If an update to a member record, when compared to the prior information for the member, scores below the overlay threshold value, then the data in the individual member has been changed to such a degree that it implies the record ID may be representing two different individuals. This also triggers an issue record for a manual check, which then can be handled through the Inspector application. This usually occurs when there is an update and the existing information is completely different from the new information, suggesting that there is an overlay of the record with another member’s information. This is represented by the red triangles below the lower threshold value.

**Note:** The Hub can be configured such that the Master Data Engine processes will result in:
• No tasks are created and only auto-links will occur
• Only tasks are created and no auto-links will occur
• Neither tasks nor auto-links will occur
• Both tasks and auto-links will occur

**Link**

The final step in the matching process is the actual linkage step. Linkages are two or more records grouped together by a common Enterprise ID. An unlimited number of records can be included in a linkage set.

There are two types of linkages:

• Automated link, generated by the Master Data Engine
• Manual link, created manually by an end-user while resolving a task

A linkage occurs by any of the following methods:

• Two records sharing a common Enterprise ID
• The score, created by a weight-based comparison of the attributes, determining an automatic linkage
• Manual instantiation regardless of score

In our simple example, the incoming record for Peter Countryman needed to be compared against three candidate records already in the Hub. The attributes of each record are passed into the comparison process—an actual comparison process would usually include many more attributes than this example—and each record is scored. Based on the limits of this example, we conclude that the only logical record from the three records related to Peter Countryman is the record for Frank Countryman, indicating a household relationship. If our threshold evaluation placed this record above the auto-link threshold, then the record for Frank Countryman would automatically link to the incoming record for Peter Countryman. The other two records—Aldo Cunningham and Sid Green—probably scored low and might require further evaluation. Hub criteria and the threshold evaluation by the Master Data Engine would create an “issue record” or would not link these records entirely. Later in this chapter we discuss “rules” which can be used to override what the Master Data Engine matching process evaluates.

**Managed task queues**

The Master Data Engine does not assume that because a record fails to score above the auto-link threshold that it should not be linked. A task might be created that facilitates user review and possible resolution. A task is a unit of work, represented by “issue records.” There are four task types.

Potential Overlay – An overlay occurs when one member’s information overrides the information of another, different, member record within the same source; thus, the same Source ID is shared by two records. An indication of this may be if a member record within a source has its attribute data change radically. Table 5 on page 18 shows an example.
Table 5. Example of Potential Overlay

<table>
<thead>
<tr>
<th>Existent record</th>
<th>Attribute</th>
<th>Value</th>
<th>Record update</th>
<th>Attribute</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Name</td>
<td>Peter Countryman</td>
<td>Name</td>
<td>Roberta Salmon</td>
<td></td>
</tr>
<tr>
<td>Phone</td>
<td>Phone</td>
<td>9119119111</td>
<td>Phone</td>
<td>602 678 9012</td>
<td></td>
</tr>
<tr>
<td>Email</td>
<td>Email</td>
<td><a href="mailto:Peter@xyz.com">Peter@xyz.com</a></td>
<td>Email</td>
<td><a href="mailto:Salmon@123.com">Salmon@123.com</a></td>
<td></td>
</tr>
<tr>
<td>Source ID</td>
<td>Source ID</td>
<td>101</td>
<td>Source ID</td>
<td>101</td>
<td></td>
</tr>
<tr>
<td>Member ID</td>
<td>Member ID</td>
<td>2345</td>
<td>Member ID</td>
<td>2345</td>
<td></td>
</tr>
</tbody>
</table>

Potential Duplicate – If two records from the same source have a score between the clerical review and auto-link thresholds, a task for Potential Duplicate is created. For example, the Regional Medical Center at Get Well Hospital has the same patient record twice under two registration numbers, as shown in Table 6.

Table 6. Example of Potential Duplicate

<table>
<thead>
<tr>
<th>Existent record</th>
<th>Attribute</th>
<th>Value</th>
<th>Record update</th>
<th>Attribute</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Name</td>
<td>Peter Countryman</td>
<td>Name</td>
<td>P Countryman</td>
<td></td>
</tr>
<tr>
<td>Phone</td>
<td>Phone</td>
<td>9119119111</td>
<td>Phone</td>
<td>9119119111</td>
<td></td>
</tr>
<tr>
<td>Email</td>
<td>Email</td>
<td><a href="mailto:Peter@xyz.com">Peter@xyz.com</a></td>
<td>Email</td>
<td><a href="mailto:Peter@xyz.com">Peter@xyz.com</a></td>
<td></td>
</tr>
<tr>
<td>Source ID</td>
<td>Source ID</td>
<td>101</td>
<td>Source ID</td>
<td>101</td>
<td></td>
</tr>
<tr>
<td>Member ID</td>
<td>Member ID</td>
<td>2345</td>
<td>Member ID</td>
<td>5457</td>
<td></td>
</tr>
</tbody>
</table>

Potential Linkage – If two records from different sources have a score between the clerical review and auto-link thresholds, then a Potential Linkage task is created. The record in the orthopedic department and the record in the dental department for Peter Countryman score are close, but the records did not score above the auto-link threshold (refer to Table 7).

Table 7. Example of Potential Linkage

<table>
<thead>
<tr>
<th>Existent record</th>
<th>Attribute</th>
<th>Value</th>
<th>Record update</th>
<th>Attribute</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Name</td>
<td>Peter Countryman</td>
<td>Name</td>
<td>P Countryman</td>
<td></td>
</tr>
<tr>
<td>Phone</td>
<td>Phone</td>
<td>9119119111</td>
<td>Phone</td>
<td>9119119111</td>
<td></td>
</tr>
<tr>
<td>Email</td>
<td>Email</td>
<td><a href="mailto:Peter@xyz.com">Peter@xyz.com</a></td>
<td>Email</td>
<td><a href="mailto:PC@xyz.com">PC@xyz.com</a></td>
<td></td>
</tr>
<tr>
<td>Source ID</td>
<td>Source ID</td>
<td>101</td>
<td>Source ID</td>
<td>102</td>
<td></td>
</tr>
<tr>
<td>Member ID</td>
<td>Member ID</td>
<td>2345</td>
<td>Member ID</td>
<td>5457</td>
<td></td>
</tr>
</tbody>
</table>

Review Identifier – Two or more records that contain the same unique identifier, for example a Social Security number, require that a user manually link them or correct the identifier if one member has an incorrect identifier.

Note: In addition to the task types discussed above, there are two system tasks: HasShadow and PreMerge. Users reviewing records and tasks will not see these two types in Initiate applications because they are used primarily by the Message-Based Transaction Services to trigger an internal system action.
**Task workflow**

The Master Data Engine can place notifications of tasks into managed work queues. Inspector for Data Resolution is a sophisticated application designed specifically for efficient resolution of task managed queues. The resolution of a task by the application user determines the state of the task.

Some example resolutions could be

- Not same person
- Merged – survivor
- Deferred – supervisor review. If users cannot resolve a task based on the information available, they can change the task status to “Deferred.” This helps in identifying the tasks that were reviewed but not resolved. It does not, however, have any impact on the task. The task rows might be deleted; the member might be promoted into a linkage or dropped out of the task set (see “Dynamic nature of tasks” below).

If a task is marked as “Resolved,” then it is automatically deleted and removed from the managed queue.

**Dynamic nature of tasks**

The dynamic nature of tasks is reflected by a number of factors. When a task record is selected, the task is dynamically formed based on the real-time comparison of that member’s data to others’ at that point in time. Task records also can be re-formed based on new or changed information—the associated score could increase or decrease, the issue record could be removed, or the issue record could be promoted to higher task type in the hierarchy. When issue records are resolved based on user input, they are removed from the queue.

When a task search is requested by an IBM Initiate application, the results are generated in real time (at the moment the search is initiated). The record, or criteria, that the Master Data Engine uses to compare against other records (candidates) is called the trigger member or trigger record (that is, the record triggering the comparison). When a task search returns results, the trigger member typically displays with a higher score than the other records. The score is higher based on the trigger record being compared against itself during the comparison process. The other records returned are issued comparison scores based upon their comparison against the trigger record. A record is also considered to be the trigger member if an update to that member causes a cross-match which results in task creation.

For example, consider record C that is assigned a Potential Duplicate task in comparison with records A and B. Any changes to C will affect the task status. Hence C is said to be the trigger record. This capability where a task could get automatically deleted or changed based on the update to the trigger record makes the tasks dynamic.

In short, a trigger record is a member record that has a record in the task table. A, B and C may all have task records, and therefore would all be considered trigger records.

**Task hierarchy**

Although a record may have numerous data issues, the Master Data Engine only allows a record to be part of a single task at one time. The assignment is based on
a task type hierarchy—some data issues are more important than others and should be resolved first. This hierarchy—from highest to lowest—is as follows:

- Potential Overlay
- Potential Duplicate
- Potential Linkage
- Review Identifier

Occasionally, updates to member data can cause a new, more serious, data issue. When this occurs, the record is promoted to the higher task type. Likewise, updates can cause the data issue to be removed, thus removing the task.

**Record promotion**

The tasks as mentioned above are listed in the order of importance (most important to least important). Whenever there is a change that involves a new issue with an existing trigger record, the record is assigned the more important of the two task types. The information about the tasks is stored in the following tables.

**Table 8. Tasks**

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mpi_memxtsk</td>
<td>Stores information about member records that have an associated task (Potential Overlay task). Details of tasks that have the value “M” in the “tskkind” column of mpi_tsktype are stored in this table.</td>
</tr>
<tr>
<td>mpi_entxtsk</td>
<td>Entity task relationship data is stored in this table (Potential Duplicate, Potential Linkage and Review Identifier tasks). Details of tasks that have the value “E” in the “tskkind” column of mpi_tsktype are stored in this table.</td>
</tr>
</tbody>
</table>

**Rules**

Rules are pair-based and are set between two members. Rules can be used to override the decisions made by the Master Data Engine during comparison. For example, there may be two records of twins and they might compare very high, but an administrator can review them and set as a rule that the two persons are never the same (non-identity rule). Therefore, even if there is an update on the records, these particular records will never be linked or even put in the review queue.

The second type of rule is the identity rule, where an administrator manually links two records. The rule then says that the two records are always the same, regardless of future data changes.

**Versioning – the life cycle of an attribute**

Refer to “Verticalization” on page 11. Each of the attribute tables (such as mpi_memphone, mpi_memident, mpi_memattr), contains a record status column called “recstat”, which determines the state of the record. The possible values for “recstat” are:

- **Active**: This status indicates that this value is the last known value to be true. (Usually corresponds to last update.)
- **Inactive**: This status indicates that this value is the previously known value to be true.
• Deleted: This status indicates that this value has been logically deleted and will
not be used any longer for comparison purposes, but is still stored to maintain
history.

• Shadow: This status indicates that this is the updated value pending approval
from a source. For example, when a user approves a manual linkage and
changes the value for the residence phone number in the new record. The
Master Data Engine marks the phone number as “Shadow” and creates a system
task that is captured by the Outbound Broker, which in turn sends the new
value to the source system. After receiving an update message from the source
that confirms the change, this flag turns to “Active” state.

**Note:** Old attribute values are not deleted but marked as Inactive. Thus, we can
learn the history of various attributes of a member over a period of time.

---

**Using dictionary data**

Dictionary data is the configuration data. The main dictionary data includes user
head, group head, segment head, source head segment attributes, and enumerated
data type elements.

**User head**

User data is the basis to secure access to the Master Data Engine and Hub. User
authentication fails if the user data is not present in the mpi_usrhead table. Note
that the Master Data Engine does not recognize new users created in LDAP until
that user first logs into the hub, whereupon the user ID is added to mpi_usrhead.

**Group head**

Group head gives a comprehensive permission set. By assigning a user to a
particular group, a security role is initiated. Administrative and application
programs may restrict access to both data and processing paths based on the list of
granted permissions.

The permissions can be granted at segment, interaction and attribute levels. These
can be defined using the mpi_grphhead, mpi_grpxseg and mpi_grpxixn tables.

**Segment head**

*Segments* coincide with the Hub database schema to define Master Data Engine
behavior and member information. Each segment is an encapsulation of a single
row from the corresponding database table. A set of fixed (primitive) segments is
packaged with the Master Data Engine. For example, MemName, MemAddr,
MemIdent are all fixed segments.

This provides a bridge between the logical names for storage segments and the
physical tables in which the segment data is stored. The mpi_seghead table
contains the information for the segments: the segcode column contains the
segment name, and the segname column contains the name of the table
corresponding to the segment.

**Source head**

This dictionary data registers identifier systems. All data defined as an identifier is
associated with a registered source. The mpi_srchead table provides the list of
registered sources. A source is given a type of ‘D’ (for definitional) when it can be
used to represent member native system sources, and ‘I’ (for informational) for any
other identifying body.
Another way to define source identifiers is that a definitional source is one in which members (records) are created and usually updated. An informational source typically provides “legal” values that can be used as attribute values. For example, an informational source may provide valid credit card numbers or frequent flyer numbers. Member records are not created in informational sources.

**Segment attributes**

Segment attributes describe the named attributes that the Master Data Engine will store using the primary data type. The mpi_segattr table contains the list of attributes that Master Data Engine will track for members.

**Enumerated data type elements**

The mpi_edtelem segment provides the standard values that an enumerated data type (EDT) recognizes. There are two styles of enumerated data translation:

- When stored as an enumerated data segment
- When stored as a reference value for a member attribute.

In the first case, the “segcode” value would be MEMENUM.

In both cases, the “edtcode” column of the mpi_segattr table should be set appropriately to the corresponding value in mpi_edtelem.

The difference in the two cases arises when we insert a value. When used as a simple lookup for a MemAttr segment, any value is accepted. When used as an enumerated data segment, the value is checked against a set of specified values for that EDT. If it does not match any of the values, it is rejected.

For example, assume there is a defined EDT called “SEX,” with only “M” and “F” in the mpi_edtelem table, using respective expansion values of “Male” or “Female” and assigned respective record numbers of 12 and 13. If a segment in mpi_segattr is defined as being a MemEnum segment, then an insert action of value “M” will actually store the number 12 and retrieval will obtain “Male”. However, an input value that is not “M” or “F” will be rejected (because there is no corresponding “elemrecno”).

On the other hand, if the mpi_segattr definition specifies a MemAttr with an “edtcode” of “SEX,” then any value is allowed to be input (and will be stored as input). Upon retrieval, the expanded value can be obtained (if an associated expansion exists).

---

**Quiz yourself: architectural and technical overview**

1. Records from the same source that possess a high score are:
   a. Candidates for Potential Overlay
   b. Duplicates
   c. Candidates for manual linking
2. When a task is promoted, that means:
   a. It is being auto-linked
   b. It needs review by a supervisor
   c. A change in data has caused the task to move up the hierarchy
3. The process of storing each attribute of a member in a separate child table is called:
a. Normalization  
b. Standardization  
c. Verticalization  

4. Two members of the same source have the same Member ID and different attributes. Then the task type is:  
a. Potential Duplicate  
b. Potential Overlay  
c. Auto-link  

5. What technique in the derivation process is used to narrow down the possible candidates when performing a search?  
a. Bucketing  
b. Pruning  
c. Branching  

Answers  
1. ‘B’ - Duplicates 
2. ‘C’ - A change in data has caused the task to move up the hierarchy 
3. ‘C’ - Verticalization 
4. ‘B’ - Potential Overlay 
5. ‘A’ - Bucketing
Chapter 2. Getting started with the SDK

The SDKs consist of a complete set of APIs that enable you to interact with the Master Data Engine software and to configure it to suit your needs. The APIs enable searching, retrieving, and editing of member and dictionary information.

The clients and applications of IBM Initiate Master Data Service, such as IBM Initiate Enterprise Viewer and IBM Initiate Inspector, use the same APIs provided for the Initiate SDKs to perform various interactions with Master Data Engine. However, you can use the SDK to build custom solutions and applications that meet your unique needs, and to include additional functionality. In this section we will discuss the various components and functionality of the SDK.

The API interaction diagram illustrates how the API interacts with the Master Data Engine through MPINET.

You can use the SDK to interact with the Master Data Engine. From basic tasks like establishing a connection to configuring its behavior and defining the comparison strategy, you can perform any interaction or get/modify data model elements.

Web services basics

Web services are the application or business logic that is provided to interact with Master Data Engine. Web services uses the Simple Object Access Protocol (SOAP) and Web Services Descriptive Language (WSDL) standards, which enable you to program in Java, C++, Microsoft .NET 2.0 or any other language that supports these standards.

Using the web services you can retrieve and modify data model elements. Some of the operations that enable you to accomplish these tasks are listed below.

- Dictionary operations – Add, update and retrieve dictionary data
- Member operations – Add, update, delete, undelete, drop, compare, search, match, merge and unmerge member data
- Task operations – Add, update, search and retrieve task data
- Enterprise Identity Assignment (EIA) operations – Add, update and search EIA data
Note: Throughout the document, web services refers to the server implementation while the web services API refers to the stubs generated from the WSDL.

Implementing web services

The service provider classes form the core of the web services. Each operation in web services has a Java class associated with it. This class contains a method, which has the business logic that realizes the task requested and is called as the service provider class. The method is referenced as the service provider method for that operation. These classes make use of the Apache Axis libraries for basic web services related tasks (for example, processing SOAP messages). The classes use the Java API to interact with the Master Data Engine.

Note: Apache Axis is both a SOAP engine and a framework for constructing SOAP processors such as clients, servers, and gateways.

The web services SDK constitutes a complete package of APIs that enable you to interact with the Master Data Engine to configure its behavior, to retrieve and modify data model elements, and to build custom applications and solutions. Figure 8 illustrates the web services, which consists of Apache Axis, service provider classes and the Java SDK, and its interaction with the Master Data Engine through MPINET.

Figure 8. Web services Architecture

The client application uses SOAP messages to interact with web services, which in turn interacts with the Master Data Engine and performs the requested operation. The SOAP messages that are created are in RPC/Literal format with out multi-refs (versions prior to 7.x used RPC/Encoded with multi-refs). While the examples use Java and Apache Axis, in theory, your clients can use what ever language or soap implementation that you choose. The clients simply need to be SOAP 1.2 and WSDL 1.1 compliant and that the following simple Java data types can be converted to the native language: String, boolean, byte, short, int, long, float, and double. For example the Microsoft .NET 2.0 framework is compatible with these types.

Refer to "compareMember web services example" on page 67 for a discussion of the SOAP message format.
Sequence of operations

The request from a client application reaches the web services in the form of a SOAP envelope. The SOAP envelope is an XML message, which contains the operation name along with the input parameters encoded as web services types. Apache Axis, which resides on the web server, processes the SOAP message and instantiates the request object. Apache Axis then uses this request object to invoke the corresponding service provider class.

The respective service provider method executes the business logic for the requested task by using the Java API to interact with the Master Data Engine.

The service provider method subsequently uses Apache Axis to generate a SOAP message from the output received from the Master Data Engine and sends the SOAP message back to the client application.

The mapping between web services types, operations, and the implementation classes is defined in the Web Services Deployment Descriptor (WSDD) file.

Enabling logging and debugging

About this task

To yield an echo of the input and output Rowlist contents in the Tomcat stdout_date log, follow these steps:

Procedure

1. From the tomcatDir/webapps/initiatews/WEB-INF/ directory, open the server-config.wsdd file.
2. Search for the line: `<parameter name="InitiateDebug" value="false"/>`
3. Set the value to true.
4. Search for the line: `<parameter name="InitiateDump" value="false"/>`
5. Set the value to true.
6. Save and close the file.
7. Restart the web services SDK Tomcat application.

Web services exposure to the clients

Web services are exposed to the clients through the WSDL, which is a specification to describe networked XML-based services. It provides a simple way for service providers to describe the basic format of requests acceptable by the service provider’s systems. Understanding the mapping of the various WSDL elements to the web services API operations and data model is the key to using web services. Our WSDL uses the RCP style with literal encoding.

IWS data model

The web services data model consists of requests and segments. There is a request type for each operation defined in the WSDL and each request type contains one or more segments. Segments are explained in detail in “Segments” on page 32.
Installing the SDK

About this task

The installation steps vary depending on the SDK you are installing. For the Web Services SDK, installation procedures are covered separately in the Web Services SDK Installation Guide.

Installing the SDK for Java

About this task

Your development environment must meet the following requirements:

- Installation of the Initiate Master Data Engine release 10.0
- Installation of the Java SDK on your system
- Enabling of network access to a supported database platform; refer to the IBM Initiate Master Data Service System Requirements sheet for certified OS and database platforms

To install the SDK for Java, follow the instructions below.

Procedure

1. Start in a working directory. Copy the IBM_Initiate_JavaSDK_10.0.0.xxx.jar file from the software distribution CD to the working directory. This cross-platform executable jar file contains the Java API and supporting libraries, as well as the HTML documentation.

2. Run the jar using the Java Platform SE binary. For example, in Windows, you can right-click the jar file in the Windows Explorer and choose Open with... > Java Platform SE binary. The install wizard opens.


4. If you agree to the terms of the license agreement, select I accept both the IBM and the non-IBM terms. and click Next.
   a. If you don't agree, click the I do not accept the terms in the license agreement. A dialog opens asking you to confirm that you are declining the license agreement.
   b. To end the installation process, click Yes on the dialog.
   c. To continue the installation process, click No. You may then choose to accept the license agreement.

5. Choose a destination folder to install the Java SDK. The default path is C:\Program Files\IBM\Initiate\JavaSDK10.0.0 Click Next.

6. Review the Pre-Installation Summary. To correct any errors, click Previous. To perform the installation, click Install. This may take a few minutes to finish.

7. Click Done to exit the installation wizard.

8. Navigate to the installation directory and create a subdirectory to hold your compiled code.

9. Add the C:\Program Files\IBM\Initiate\JavaSDK10.0.0\madapi.jar file to the Java CLASSPATH environment variable.

Results

You are ready to use the SDK for Java.
Establishing a connection

About this task

To interact with the Master Data Engine programmatically using the JavaSDK, you need to establish an MPINET connection. You can accomplish this with the Context object. This class maintains a TCP/IP socket connection with the Engine and provides methods for authentication and disconnecting. Connections can optionally be secured via Secure Sockets Layer (SSL). Programmatically it is best to minimize the amount of times a Context is created and destroyed. Context objects should be created and then re-used for best performance. If a Context is to be used by more than one thread concurrently you should implement one of the following solutions:

- Control access to it through synchronization
- Create a Context for each thread performing an interaction
- Create a pool of Contexts for sharing between threads that can be checked in and out

Making MPINET connections with Java

About this task

You can use the following Context constructor to establish an MPINET connection with the Master Data Engine:

```
Context(java.lang.String hostName, int hostPort)
```

where:

- `hostName` = The hostname of the server running the MPINET service
- `hostPort` = The port on which the MPINET service is listening

For example:

```
Context ctx = new Context("mymachine", 15000);
```

This will establish an MPINET connection to the Master Data Engine running on mymachine and listening on port 15000. The host name string “mymachine” could be an IP address or a server name.

**Note:** Please consult the 10.0 Java API HTML documentation for additional functionality contained in the Context object.

Making Java MPINET connections over SSL

The SDK for Java can be configured to encrypt communication with the Master Data Engine SSL. The Master Data Engine must also be configured for SSL communication on the specified host and port.

About this task

Use the following Context constructor to establish an SSL enabled MPINET connection with the Master Data Engine:

```
Context(java.lang.String hostName, int hostPort, java.util.Properties sslProps)
```

where:

- `hostName` = The hostname of the server running the MPINET service
hostPort = The port on which the MPINET service is listening

sslProps = Set of SSL related options

The following options must be set in the sslProps to enable SSL:

- MpiNetSecure.SECLIB - Security library option used to enable SSL
communication between the web services SDK and the Master Data Engine. Set
to “SSL” to enable.
- MpiNetSecure.SSLVERSION - SSL version option must match version specified
  in the Master Data Engine, usually “SSLv3”.
- MpiNetSecure.SSLCERTVERIFY - Option for the Java SDK client to verify the
  SSL certificate presented by the Master Data Engine. Set to “true” to enable
certificate verification. If enabled, the Server’s SSL Certificate must be trusted by
the Java SDK as described below. If not enabled, the Server’s SSL Certificate is
just used for encryption purposes and not verified by the Java SDK.

For example:

```java
Properties sslProps = new Properties();
sslProps.setProperty(MpiNetSecure.SECLIB, "SSL");
sslProps.setProperty(MpiNetSecure.SSLVERSION, "SSLv3");
sslProps.setProperty(MpiNetSecure.SSLCERTVERIFY, "false");
Context ctx = new Context("mymachine", 15000, sslProps);
```

This establishes an MPINET connection to the SSL enabled Master Data Engine
instance running on mymachine and listening on port 15000. The communication
will be encrypted, but the client will not verify the Server’s SSL certificate.

In addition to the sslProps, the keyStore for the Java VM used by the Java SDK
client must be configured with the proper SSL certificates if the
MpiNetSecure.SSLCERTVERIFY option is enabled. Java uses a default keyStore to
verify the Server’s SSL certificate if no additional configuration is provided. If the
Server’s SSL certificate is signed by a trusted Certificate Authority known to the
Java SDK, then no additional configuration is required. If the Server’s certificate
uses a custom certificate or internal CA, then the default Java keyStore can be
updated to include information about the certificate (refer to the JSSE Reference
Guide noted below for information on updating the default Java keyStore). If the
default keyStore file is not used, the following VM options may need to be
specified when starting the Java SDK client process.

- -Djavax.net.ssl.keyStore = Full path to the custom keyStore file
- -Djavax.net.ssl.keyStorePassword = KeyStore password
- -Djavax.net.ssl.keyStoreType = KeyStore type

For more information on using SSL with Java, refer to the JSSE Reference Guide for
the JDK located at

[http://java.sun.com/j2se/1.5.0/docs/guide/security/jsse/JSSERefGuide.html](http://java.sun.com/j2se/1.5.0/docs/guide/security/jsse/JSSERefGuide.html)

**Disabling Java MPINET connection KEEPALIVE**

By default the Java SDK enables the KEEPALIVE option for all socket connections
to the Master Data Engine. Every two hours the Java SDK sends a KEEPALIVE
packet to the engine to verify that the connection exists and to keep the connection
with the engine alive. To disable KEEPALIVE, the connection properties must be
modified.
About this task

Use this Context constructor to establish an MPINET connection with the engine.

```java
Context(java.lang.String hostName, int hostPort, java.util.Properties sockProps)
```

where

- `hostName` = The hostname of the server running the MPINET
- `service hostPort` = The port on which the MPINET service is listening
- `sockProps` = Set of socket-related options

In the `sockProps` property, set the `MpiNet.KEEPALIVE` option to false.

For example

```java
Properties sockProps = new Properties();
sslProps.setProperty(MpiNet.KEEPALIVE, "false");
Context ctx = new Context("mymachine", 15000, sslProps);
```

This setting establishes an MPINET connection to the engine instance running on `mymachine` and listening on port `15000`. After setting `MpiNet.KEEPALIVE` to false, the Java SDK no longer sends KEEPALIVE packets to the engine. The communication will be severed if your firewall is configured to sever stale connections.

Workflow of an interaction using the SDK for Java

The MemRowList objects are used both to provide input parameters and to receive output data from the Master Data Engine.

Before going into the intricacies of SDK APIs, let us understand the steps involved in adding a member (MemPut).

1. Establish an MPINET connection with the Master Data Engine by creating a Context Object.
2. We construct the Input parameters through segments like MemHead and MemName. Each one of these segments represents a row. Each row contains different attributes that are used to make up a member. Each row is then added to a MemRowList and passed in as a parameter to the MemPut Interaction. The MemPut interaction sends these attributes to the Master Data Engine where the rows are processed and relevant tables in the Master Data Engine database are either inserted or updated.
3. Output from the Master Data Engine is also a MemRowList, which is again a collection of rows that represent one or more member objects. You can retrieve members by executing the MemGet Interaction, which also takes a MemRowList as a parameter.

**Note:** Read more about RowLists in "RowList objects" on page 40.

### Segments

Segments are the building blocks of the IBM Initiate Master Data Service software. They coincide with the data source schema to define the Master Data Engine behavior and member information. Each segment class is an encapsulation of a single row of the coinciding Hub database table.

Web services users: Each WSDL complexType is mapped to a Java bean with its fields as the elements of the complexType.

### Representation of MemName segment

The following figure depicts the representation of a MemName segment. The mpi_memname table stores the data for the MemName class. The MemName segment class is a subclass of MemAttrRow, which is a subclass of MemRow, which is a subclass of the Row class.

**Table 9. MemName Segment**

<table>
<thead>
<tr>
<th>Segment name</th>
<th>Hub Database table name</th>
<th>SDK class name</th>
<th>Segment attribute code</th>
</tr>
</thead>
<tbody>
<tr>
<td>MemName</td>
<td>mpi_memname</td>
<td>MemName</td>
<td>GTNAME</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>LGLNAME</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>CUSTNAME</td>
</tr>
</tbody>
</table>

The MemName segment can contain many different types of names; these names are differentiated by the segment attribute code such as GTNAME (Guarantor Name), LGLNAME (Legal Name), and CUSTNAME (Guest Name).

Segments can be divided into two main groups:
Dictionary segments provide type definition and lookup values. These values define customer specific data, Master Data Engine behavior and other segment types.

Member segments enable you to define individual attributes for a member, inform you of linkages between members and alert you of task between members.

There are two types of member segments: fixed (or primitive) and implementation-defined segments (IDS). Fixed segments are those that are created and shipped with the API. MemName, MemAddr, and MemIdent are all examples of fixed segments. All member segments available in previous releases are now called fixed segments. Implementation-defined segments are segments that are defined during implementation of the IBM Initiate Master Data Service software at a client site and, therefore, could not have a generated class associated with it. Using implementation-defined segments are explained later in this document.

Each segment group is explained further in the following sections.

**Dictionary segments**

You can use the dictionary segments to provide the Master Data Engine with the lookup values. These lookup values define customer-specific data and provide Master Data Engine behavior criteria. Dictionary segments can be broken down further into the following categories.

**Type definition segments**

Type definition segments provide lookup values that are used to define data types used by the Master Data Engine.

For example:

*Table 10. Type Definitions*

<table>
<thead>
<tr>
<th>Segment</th>
<th>Database table</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AppHead</td>
<td>mpi_apphead</td>
<td>Application property header. Contains the name of applications interacting with Master Data Engine software.</td>
</tr>
<tr>
<td>AppProp</td>
<td>mpi_appprop</td>
<td>Contains property definitions for applications interacting with Master Data Engine software.</td>
</tr>
<tr>
<td>AppData</td>
<td>mpi_appdata</td>
<td>Defines application configuration data. Enables applications to store data at the database level.</td>
</tr>
<tr>
<td>EdtElem</td>
<td>mpi_edtelem</td>
<td>Enumerated data type defined elements</td>
</tr>
<tr>
<td>EdtHead</td>
<td>mpi_edthead</td>
<td>Enumerated data type header</td>
</tr>
<tr>
<td>EiaStat</td>
<td>mpi_eiastat</td>
<td>Enterprise ID assignment status. Includes Unexamined, Examined-OK, and Examined-Error.</td>
</tr>
<tr>
<td>EiaType</td>
<td>mpi_eiatype</td>
<td>Enterprise ID relationship type. Includes auto-link, manual-link, merge, unmerge, delete, undelete, and premerge.</td>
</tr>
<tr>
<td>EntType</td>
<td>mpi_enttype</td>
<td>Entity type definitions. Defines valid entity types. Entities can utilize different comparison algorithms and weight tables.</td>
</tr>
</tbody>
</table>

Chapter 2. Getting started with the SDK 33
Table 10. Type Definitions (continued)

<table>
<thead>
<tr>
<th>Segment</th>
<th>Database table</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EvtType</td>
<td>mpi_evttype</td>
<td>Event type management. Includes add, change, merge, and delete. If a database interaction request is event-oriented, and the identifying interaction type is not listed in the MPI_evttype table, Master Data Engine software rejects any further processing of the request.</td>
</tr>
<tr>
<td>Handler</td>
<td>mpi_handler</td>
<td>Callback handler segment. Indicates what handlers should be invoked for any given Callback Type and IxnCode combination.</td>
</tr>
<tr>
<td>MemStat</td>
<td>mpi_memstat</td>
<td>Member status management</td>
</tr>
<tr>
<td>MemType</td>
<td>mpi_memtype</td>
<td>Member type management. Defines the standardization and derivation routines for each member type.</td>
</tr>
<tr>
<td>TskStat</td>
<td>mpi_tskstat</td>
<td>Task status or record state (workflow status)</td>
</tr>
<tr>
<td>TskType</td>
<td>mpi_tsktype</td>
<td>Task type. Includes potential overlay, potential duplicate, potential linkage, review identifier, premerge, and hasShadow.</td>
</tr>
</tbody>
</table>

MemType segment coincides with the database table mpi_memtype, which defines four standard member types: Person, Guest (or Customer), Provider and Organization.

EntType segment coincides with the database table mpi_enttype, which defines two standard entity types: Identity and Household.

**Segment definition segments**

These segments contain the definition or the shape of the Master Data Engine fixed segments. The definition segments also enable you to define customer-specific implementation-defined segments that will be used by the Master Data Engine.

**Note:** IBM Initiate Workbench is a powerful and easy-to-use application that enables the user to create each implementation-defined segment. The application will then automatically create all the rows necessary for the implementation-defined segment in the segment definition tables.

For example:

Table 11. Segment Definitions

<table>
<thead>
<tr>
<th>Segment</th>
<th>Database table</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SegHead</td>
<td>mpi_seghead</td>
<td>Segment header information</td>
</tr>
<tr>
<td>SegAttr</td>
<td>mpi_segattr</td>
<td>Segment attributes</td>
</tr>
<tr>
<td>SegXfld</td>
<td>mpi_segxfld</td>
<td>Field definitions contained in the segment</td>
</tr>
<tr>
<td>SrcAttr</td>
<td>mpi_srcattr</td>
<td>Attribute definitions by Source.</td>
</tr>
</tbody>
</table>

SegHead segment coincides with the database table mpi_seghead, which defines all internal segments and their segment code (SegCode). Fixed segments such as MemName, MemAttr, and MemIdent are defined in SegHead.
SegAttr segment coincides with the database table mpi_segattr. SegAttr rows contain the customer specific segments definitions for each SegHead. You can also have multiple SegAttr definitions for each SegHead definition. For example the SegCode MemPhone could be defined to have an Attribute Code (AttrCode) of HOMEPHON and of WORKPHON. This allows you to store two types of phones in the same SegHead’s SegCode but have them uniquely identified as different types of phones by the use of the SegAttr’s AttrCode.

SegXfld segment coincides with the database table mpi_segxfld. SegXfld rows define all the field names and data types for all the fields contained in a segment.

SrcAttr segments define what the valid sources are for each attribute.

**Note:** The “AttrCode” found in the mpi_segattr table is the value used on the ‘setAttrCode’ method when creating or working with any MemAttrRows.

The presence of an “EdtCode” in the mpi_segAttr table indicates the mpi_edtelem table can be retrieved and used to provide values for such attributes as drop-down lists, radio buttons, and list box selections.

**Source definition segments**

Source definition segments enable you to define recognized source systems for the Master Data Engine and how the source systems will interact. You can provide descriptive information about the source system through this segment.

For example:

<table>
<thead>
<tr>
<th>Table 12. Source Definitions</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Segment</strong></td>
</tr>
<tr>
<td>SrcHead</td>
</tr>
<tr>
<td>SrcXsrc</td>
</tr>
</tbody>
</table>

SrcHead segment coincides with the database table mpi_srchead, which defines all source names with their source code (SrcCode). Sources that are defined in the mpi_srchead table with a SrcType of “D” are Definitional and typically represent a source system that is a source of member data.

Member identification data, such as Social Security number or driver’s license, is modeled by the mpi_memident table. The mpi_memident table contains an identification number and a reference to the identifier issuer. This identifier issuer is a source that is defined in mpi_srchead table with a SrcType of “I.” When creating MemIdent rows the “idsrcrcrecno” column in the mpi_memident table can be mapped on to the “srcrecno” column of the mpi_srchead table to get the identifier issuer details.

**Use segments**

The Use segments provide the specific configuration that defines Master Data Engine behavior. Here we define the strategies for comparing the attributes, derivation of data, standardization of data, the rules for linkages.

For example:
<table>
<thead>
<tr>
<th>Segment</th>
<th>Database table</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BktFunc</td>
<td>mpi_bktfunc</td>
<td>Defines bucket functions, such as stdfuncs, dvdfuncs, and cmpfuncs</td>
</tr>
<tr>
<td>BktXgen</td>
<td>mpi_bktxgen</td>
<td>Defines bucket generation types per bucket function</td>
</tr>
<tr>
<td>CmpHead</td>
<td>mpi_cmphead</td>
<td>Comparison strategy definition</td>
</tr>
<tr>
<td>CmpSpec</td>
<td>mpi_cmpspec</td>
<td>Comparison strategy specification</td>
</tr>
<tr>
<td>CmpFunc</td>
<td>mpi_cmpfunc</td>
<td>Comparison function definitions</td>
</tr>
<tr>
<td>CvwHead</td>
<td>mpi_cvwhead</td>
<td>Defines a composite view</td>
</tr>
<tr>
<td>CvwXseg</td>
<td>mpi_cvwxseg</td>
<td>Defines segments that belong to a composite view</td>
</tr>
<tr>
<td>DvdHead</td>
<td>mpi_dvdhead</td>
<td>Derived data strategy header</td>
</tr>
<tr>
<td>DvdXbkt</td>
<td>mpi_dvdxbkt</td>
<td>Defines bucket data generation</td>
</tr>
<tr>
<td>DvdXcmp</td>
<td>mpi_dvdxcmp</td>
<td>Derived data comparison definitions. Relates the derived data code to the comparison code; relates data to a standardization role.</td>
</tr>
<tr>
<td>DvdXqry</td>
<td>mpi_dvdxqry</td>
<td>Defines query data generation. Associates a derived data element to a query element.</td>
</tr>
<tr>
<td>DvdXstd</td>
<td>mpi_dvdxstd</td>
<td>Defines standardization data generation. Associates member attributes to a standardization function to create a derived data element.</td>
</tr>
<tr>
<td>DvdYbkt</td>
<td>mpi_dvdybkt</td>
<td>Defines bucket detail data generation. Associates comparison elements to bucket elements and assigns the elements to a bucket group and bucket function.</td>
</tr>
<tr>
<td>IxnHead</td>
<td>mpi_ixnhead</td>
<td>Provides information about each interaction</td>
</tr>
<tr>
<td>LibHead</td>
<td>mpi_libhead</td>
<td>External library header</td>
</tr>
<tr>
<td>StrHead</td>
<td>mpi_strhead</td>
<td>String header</td>
</tr>
<tr>
<td>StrAnon</td>
<td>mpi_stranon</td>
<td>Anonymous string values</td>
</tr>
<tr>
<td>StrCmap</td>
<td>mpi_strcmap</td>
<td>Defines character mapping for use in multi-language support.</td>
</tr>
<tr>
<td>StdType</td>
<td>mpi_stdtype</td>
<td>Standardization data type management</td>
</tr>
<tr>
<td>StrEdit</td>
<td>mpi_stredit</td>
<td>String table edit patterns for standardization.</td>
</tr>
<tr>
<td>StrFreq</td>
<td>mpi_strfreq</td>
<td>String word frequencies</td>
</tr>
<tr>
<td>StrNbkt</td>
<td>mpi_strnbkt</td>
<td>Defines generic numeric range based bucketing. Enables generic numeric range based bucketing, (e.g., height or weight ranges).</td>
</tr>
<tr>
<td>StrShkt</td>
<td>mpi_strshkt</td>
<td>Defines generic string range based bucketing</td>
</tr>
<tr>
<td>StrWord</td>
<td>mpi_strword</td>
<td>String word classifications</td>
</tr>
<tr>
<td>StrXstr</td>
<td>mpi_strxstr</td>
<td>String table cross match table</td>
</tr>
<tr>
<td>SrcXent</td>
<td>mpi_srcxent</td>
<td>Defines SrcRecno to EntTypeno mapping for enhanced entity management.</td>
</tr>
<tr>
<td>StrEqui</td>
<td>mpi_strequi</td>
<td>Equivalent string values</td>
</tr>
<tr>
<td>SeqGen</td>
<td>mpi_seqgen</td>
<td>Unique sequence number generator</td>
</tr>
<tr>
<td>StdFunc</td>
<td>mpi_stdfunc</td>
<td>Standardization strategy function</td>
</tr>
<tr>
<td>SysKey</td>
<td>mpi_syskey</td>
<td>System registry information</td>
</tr>
<tr>
<td>SysProp</td>
<td>mpi_sysprop</td>
<td>System properties</td>
</tr>
</tbody>
</table>
Table 13. Use Definitions (continued)

<table>
<thead>
<tr>
<th>Segment</th>
<th>Database table</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wgt1Dim</td>
<td>mpi_wgt1dim</td>
<td>Defines one dimensional weight</td>
</tr>
<tr>
<td>Wgt2Dim</td>
<td>mpi_wgt2dim</td>
<td>Defines two dimensional weight</td>
</tr>
<tr>
<td>Wgt3Dim</td>
<td>mpi_wgt3dim</td>
<td>Defines three dimensional weight</td>
</tr>
<tr>
<td>Wgt4Dim</td>
<td>mpi_wgt4dim</td>
<td>Defines four dimensional weight</td>
</tr>
<tr>
<td>WgtHead</td>
<td>mpi_wgthead</td>
<td>Weight definition information</td>
</tr>
<tr>
<td>WgtNval</td>
<td>mpi_wgtnval</td>
<td>Weight definition table for comparisons</td>
</tr>
<tr>
<td>WgtSval</td>
<td>mpi_wgtsval</td>
<td>Weight definition table for comparisons</td>
</tr>
<tr>
<td>WgtType</td>
<td>mpi_wgttype</td>
<td>Weight type management</td>
</tr>
<tr>
<td>WgtXwgt</td>
<td>mpi_wgtxwgt</td>
<td>Weight table cross match table</td>
</tr>
</tbody>
</table>

CvwHead segment coincides with the database table mpi_cvwhead, which provides composite views of data across multiple segments. This allows user-defined viewing of data.

**Note:** Except for the CvwHead segment, the other Use segments are rarely used in typical applications except for administration-type programs.

**User access definition segments**

User access definition segments define the users of a given application or system and their associated access control rights. Here we create various users (with User IDs) who would be allowed to interact with the Master Data Engine to perform various tasks and roles.

For example:

Table 14. User Access Definitions

<table>
<thead>
<tr>
<th>Segment</th>
<th>Database table</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UsrHead</td>
<td>mpi_usrhead</td>
<td>User header information</td>
</tr>
<tr>
<td>GrpHead</td>
<td>mpi_grphead</td>
<td>Group header information</td>
</tr>
<tr>
<td>GrpXixn</td>
<td>mpi_grpixn</td>
<td>Group-to-interaction permissions; defines the interactions members of a defined group are allowed to perform.</td>
</tr>
<tr>
<td>GrpXseg</td>
<td>mpi_grpxseg</td>
<td>Group-to-segment definition; defines the allowable segments for a group.</td>
</tr>
<tr>
<td>IxnHead</td>
<td>mpi_ixnhead</td>
<td>Interaction header information</td>
</tr>
<tr>
<td>GrpXcvw</td>
<td>mpi_grpxcvw</td>
<td>Group composite view definition; defines the default composite view for a group.</td>
</tr>
</tbody>
</table>

UsrHead segment coincides with the database table mpi_usrhead, which defines the valid users for IBM Initiate Master Data Service components. (Note that the Master Data Engine does not recognize new users created in LDAP until that user first logs into the hub, whereupon the user ID is added to mpi_usrhead.)

**Relationship segment definitions**

Relationship segments enable the definition and management of relationships between member records in the Master Data Engine. These segments define the
types of relationships you manage and the rules used for creating relationship
linkages and tasks, and provide specific information about relationships with
associated tasks.

<table>
<thead>
<tr>
<th>Table 15. Relationship Segment Definitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Segment</td>
</tr>
<tr>
<td>--------</td>
</tr>
<tr>
<td>RelLink mpi_rellink</td>
</tr>
<tr>
<td>RelRule mpi_relrule</td>
</tr>
<tr>
<td>RelSegAttr mpi_resegattr</td>
</tr>
<tr>
<td>RelType mpi_relttype</td>
</tr>
<tr>
<td>RelXtsk mpi_relxtsk</td>
</tr>
</tbody>
</table>

**Member segments**

Member segments enable you to define individual attributes for a member, inform
you of linkages between members, and alert you of tasks between members. For example:

<table>
<thead>
<tr>
<th>Table 16. Member Segments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Segment</td>
</tr>
<tr>
<td>--------</td>
</tr>
<tr>
<td>MemHead</td>
</tr>
<tr>
<td>MemAddr</td>
</tr>
<tr>
<td>MemAppt</td>
</tr>
<tr>
<td>MemAttr</td>
</tr>
<tr>
<td>MemCont</td>
</tr>
<tr>
<td>MemDate</td>
</tr>
<tr>
<td>MemDrug</td>
</tr>
<tr>
<td>MemElig</td>
</tr>
<tr>
<td>MemEnum</td>
</tr>
<tr>
<td>MemExtA</td>
</tr>
<tr>
<td>Segment</td>
</tr>
<tr>
<td>-------------</td>
</tr>
<tr>
<td>MemExtB</td>
</tr>
<tr>
<td>MemExtC</td>
</tr>
<tr>
<td>MemExtD</td>
</tr>
<tr>
<td>MemExtE</td>
</tr>
<tr>
<td>MemIdent</td>
</tr>
<tr>
<td>MemName</td>
</tr>
<tr>
<td>MemNote</td>
</tr>
<tr>
<td>MemPhone</td>
</tr>
<tr>
<td>MemCmpd</td>
</tr>
<tr>
<td>MemBktd</td>
</tr>
<tr>
<td>MemRule</td>
</tr>
<tr>
<td>MemQryd</td>
</tr>
<tr>
<td>MemXtsk</td>
</tr>
<tr>
<td>MemXeia</td>
</tr>
<tr>
<td>EntXtsk</td>
</tr>
<tr>
<td>EntXeia</td>
</tr>
<tr>
<td>EntLink</td>
</tr>
</tbody>
</table>
Table 16. Member Segments (continued)

<table>
<thead>
<tr>
<th>Segment</th>
<th>Database table</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EntNote</td>
<td>mpi_entnote</td>
<td>Notes® for an entity</td>
</tr>
<tr>
<td>EntRule</td>
<td>mpi_entrule</td>
<td>Rules for each entity. Defines identity and non-identity rules between members and has an entity identifier appended; for example ENTRULE_Identity or ENTRULE_Household.</td>
</tr>
</tbody>
</table>

MemHead segment coincides with the database table mpi_memhead, which is the base table and contains the primary identifier. All other member segments coincide with child tables like mpi_memattr, mpi_memdate, and mpi_memname.

Attributes like name, phone, address, and email for a member will be stored in child tables.

EntLink segment coincides with the database table mpi_entlink_*, which defines the members that are linked and comprise an entity.

**Note:** The above asterisk (*) represents the enttype defined in the mpi_enttype table. For example, mpi_entlink_id or mpi_entlink_hh.

EntNote is not used in any of the existing IBM Initiate Master Data Service components.

**RowList objects**

RowList is a container class for a set of Row objects. The class is similar to a Vector, but RowLists will only store objects that are of a type derived from Row. They are mainly used as input and output parameters to interactions. Interactions are described in the following session. There are two main RowList derived objects.

- DicRowList – Used to contain dictionary segments
- MemRowList – Used to contain member segments

**RowList structure**

Understanding the structure of the RowList and its Rows is vital to the understanding of the SDK.

A MemRowList consists of one or more MemRow objects; each MemRow is a member segment. There is no one-to-one mapping between a member and a row; a member might have many rows and more than one member can be contained in the MemRowList. Consider the MemRowList listed in the following figure. For each member, there will be one MemHead and zero or more MemRow segments (specifically in this example, the MemName and/or MemDate segments).
One important aspect to remember while iterating through the MemRowList is that there is no significance in the order in which they are returned from the interactions. For example, there can be three members’ information stored in the row list, but on retrieval all the MemHead rows might be retrieved as the first three rows. On a subsequent retrieval the MemHead rows might be the last three rows or interspersed between other member segments. If order is important, the APIs provide methods found on RowList that allow for sorting and filtering of rows (see HTML examples for RowList.rows Testable and CompareKey).

Example: adding rows to a RowList

In the following example, we create a MemRowList, then add two members with MemName and MemAttr segments to each member:

```java
// Create a member row list.
MemRowList memberRows = new MemRowList();

// Create a MemHead object.
// MemHead models Initiate
// database table mpi_memhead.
MemHead memHead = new MemHead();

// Set identifier data - srcCode/memIdnum
memHead.setSrcCode("RMC");
memHead.setMemIdnum("1002423");

// Set attributes for the member.
MemAttr memAttr = new MemAttr();
memAttr.setMemRecno(memHead.getMemRecno());
memAttr.setEntRecno(memHead.getEntRecno());
memAttr.setMemSeqno(memHead.generateNextMemSeqno());

// Set the segment attribute code(Listed in mpi_segattr table)
memAttr.setAttrCode("RACE");
memAttr.setAttrVal("01");
memAttr.setRowInd(RowInd.INSERT);

// As an alternate to the above method you can use IDS methods
```
// on the Row Object to also create a MemAttr and set values to it.
MemRow mattrs = new MemRow("MemAttr");

try {
    // init row with MemRecno/EntRecno of memHead
    mattrs.init(memHead);
    mattrs.setString("AttrCode", "SEX");
    mattrs.setString("AttrVal", "M");
    mattrs.setEnumRowInd("RowInd", RowInd.INSERT);
} catch (SetterException e) {
    System.err.println(e.getMessage());
}

// Set the name of the member
MemName memName = new MemName(memHead);

// Set the segment attribute code
memName.setAttrCode("LGLNAME");
memName.setOnmFirst("Rick");
memName.setOnmLast("Jones");
memName.setEnumRowInd(RowInd.INSERT);

// Add this member information to the member row list
memberRows.addRow(memHead);
memberRows.addRow(memAttr);
memberRows.addRow(mattrs);
memberRows.addRow(memName);

// Create another MemHead object
MemHead memHead1 = new MemHead();

// Set identifier data of second member – srcCode/memIdnum
memHead1.setSrcCode("RMC");
memHead1.setMemIdnum("1002424");

// Set attributes for the second member
MemAttr memAttr1 = new MemAttr();
memAttr1.setMemRecno(memHead1.getMemRecno());
memAttr1.setEntRecno(memHead1.getEntRecno());
memAttr1.setMemSeqno(memHead1.generateNextMemSeqno());
memAttr1.setAttrCode("SEX");
memAttr1.setAttrVal("F");
memAttr1.setEnumRowInd(RowInd.INSERT);

// Set the name of the second member.
MemName memName1 = new MemName(memHead1);

// Set the segment attribute code.
memName1.setAttrCode("LGLNAME");
memName1.setOnmFirst("Shane");
memName1.setOnmLast("Kau Lee");
memName1.setEnumRowInd(RowInd.INSERT);

// Add the second member information to the row list.
memberRows.addRow(memHead1);
memberRows.addRow(memAttr1);
memberRows.addRow(memName1);

A dump of the RowList looks like:
** Mem Dump:
class madison.mpi.MemHead rowInd: A memRecno: -294967294 entRecno:
-294967294 memSeqno: 2 srcRecno: 0 srcCode: RMC memIdnum: 1002423
matchScore: 0 memStat: A srcFtime: Mon Dec 22 19:58:09 GMT+05:30 2003
Example: removing rows from a RowList

Rows can be removed from a RowList using the following method:

```java
// Remove the current row
inpMemRows.removeRow(memHead);
// Remove all rows
inpMemRows.removeAllRows();
```

Note: When you destroy a RowList, the RowList deletes the items that have been added to it.

Example: traversing a RowList

A RowList can be iterated using the FindRow and Rows methods. For example, this code block will iterate through the Row segments contained in the MemRowList object.

```java
RowIterator rowIter = outMemRows.rows();
while (rowIter.hasMoreRows())
{
    MemRow memRow = (MemRow) rowIter.nextRow();
    if (memRow instanceof MemAttrRow)
    {
        MemAttrRow memAttrRow = (MemAttrRow) memRow;
        System.out.print(" attrCode: " + memAttrRow.getAttrCode());
        if (memRow instanceof MemAttr)
        {
            MemAttr memAttr = (MemAttr) memRow;
            System.out.print(" attrVal: " + memAttr.getAttrVal());
        }
    }
    else if (memRow instanceof MemName)
```
Implementation-defined segments

Implementation-defined segments (IDS) enable customers to define custom segments to meet their specific business needs. Because these segments are defined at the time of implementation they cannot have a specific generated Java class or WSDL definition associated with it. The Row object or MemIdsWs object provides the methods that allow you to create and access IDS. The IDS Row and MemIdsWs methods are conceptually similar to the Java introspection methods.

Fixed segments are those that are created and shipped with the API. For example MemName, MemAddr, MemIdent are all fixed segments. The same methods available on the Java Row object and designed for IDS can also be used for fixed segments.

Note: Java users: The implementation-defined segment methods available in madison.mpi.Row require a valid Context to be created prior to using any of the Row methods.

Example: creating implementation-defined segments using the SDK for Java

This code block creates the fixed segments MemHead and MemName, and an IDS MemCust using the IDS methods available on the Row object. You will notice that for the fixed segments the setString() name parameter is the same as the set method name on the MemName class. For example MemName has a setOnmFirst() method, the equivalent set method using IDS is setString("OnmFirst", "Mario").

```java
//Must create your ctx first before using any IDS methods in the Row class.
Context ctx = new Context(args[0], Integer.parseInt(args[1]), usr);
MemRow memHead = null;
MemRow mn = null;
MemRow mc = null;
MemRowList inpMemList = new MemRowList();
try {
    // Create a new MemHead.
    memHead = new MemRow("MemHead");
    //init row with temp MemRecno/EntRecno by setting to 0
    memHead.init(0, 0);
    //Set String Values for SrcCode and MemIdnum
    memHead.setString("SrcCode", "RMC");
    memHead.setString("MemIdnum", "112233");
    // Create a MemName and use IDS Row
    // methods to setString values that equate to
    // the primitives segment set methods.
    mn = new MemRow("MemName");
    // Give it the same MemRecno/EntRecno as the MemHead
    mn.init(memHead);
    mn.setString("LGLNAME");
    mn.setString("OnmLast", "da Tester");
    mn.setString("OnmFirst", "Mario");
    // Create a MemCust IDS Row
```
mc = new MemRow("MemCust");
// Give it the same MemRecno/EntRecno as the MemHead
mc.init(memHead);
mc.setString("AttrCode", "CUST");
// For when you have your data values as all Strings
// but the data type is a non-string value
// you can use setAsString()
// and the correct conversion will occur, such as:
// mc.setAsString("custScore", "20");
// The above is the equivalent to:
mc.setShort("custScore", (short)20);
mc.setInt("custRecno", 202032);
// setAsString will convert this String into a Date if it
// is in a supported format as defined in madison.util.ConvDate
mc.setAsString("custDate", "2005-05-20 10:14:43");
mc.setAsString("custval", "val99");
}
// SetterException is thrown if the value can not be set.
catch (SetterException ex)
{
    System.err.println("SetterException: "+ex.getMessage());
    System.exit(0);
}
inpMemList.addRow(memHead);
inpMemList.addRow(mn);
inpMemList.addRow(mc);

Example: creating implementation-defined segments using web services

This code block creates the fixed segment MemHead and an IDS MemCust segment using the MemIdsWs segment. Where the shape or contents of IDS segments have to be dynamically learned the SegDefWs and FldDefWs segments can be used. These segments contain all the IDS object definitions. The following example creates three MemIdsWs segments demonstrating three different techniques for creating an IDS and setting data values for the segment.

//create member object that will hold rows to add
Member member = new Member();

//Create primitive MemHeadWs for a Member to put
MemHeadWs memHead = new MemHeadWs();
memHead.setSrcCode(m_sSrcCode);
memHead.setMemIdnum(m_sMemIdnum);
member.setMemHead(memHead);

//Create MemIdsWs MEMCUST segment for a Member put
//Assumes a Member Custom (MEMCUST) segment is defined in the Hub.
//MEMCUST contains the fields custval, custdate, custscore, curstrecno
MemIdsWs[] memIds = new MemIdsWs[3];
memIds[0] = new MemIdsWs();
memIds[0].setSegCode("MEMCUST");
memIds[0].setAttrCode("CUST");

//You must set the FldNames in the same order as
// the setValues() method will be in.
memIds[0].setFldNames(new String[] {"custval", "custdate"});
memIds[0].setValues(new String[] {"value2", "2005-5-2 1:04:43 PM"});

//Create 2nd row for MEMCUST
memIds[1] = new MemIdsWs();
memIds[1].setSegCode("MEMCUST");
memIds[1].setAttrCode("CUST");

//This displays another technique for setting values,
//because the order of the data matches the default fldName order.
//using the setFldNames method is not necessary.
memIds[1].setValues(new String[] {"value6", ",", ",", ","});

//The following usage would be an invalid setValue()
//as the array does not match the number of FldNames defined
//for MEMCUST and setFldNames() was not used.
//memIds[1].setValues(new String[] {"sco6Value", ","});

//Create 3rd row for MEMCUST
memIds[2] = new MemIdsWs();
memIds[2].setSegCode("MEMCUST");
memIds[2].setAttrCode("CUST");

//This shows a unique way of setting data. The setting of 2 custvals is
//allowed but only the last data value or "value5" in this example will be set.
//For "custval" first "value4" will be set but then will be replaced with "value5".
//The end result is "custval=value5".
memIds[2].setFldNames(new String[] {"custscore", "custdate", "custrecno", "custval",
                                   "custval"});
                                        "value5"});
member.setMemIds(memIds);

Example: getting data from implementation-defined segments
using Java

This code block gets data from the fixed segments MemHead, MemName and
MemAttr using the IDS methods available on the Row object. You will notice that
for the fixed segments the getString() name parameter is the same as the get
method name on the MemName class. For example MemName has a
getOnmFirst() method, the equivalent get method using IDS is
getString("OnmFirst").

try
{
//This is the IDS equivalent of using instanceof
if (memRow.getSegCode().equals("MemHead"))
{
    System.out.print(" srcCode : " + memRow.getString("srcCode"));
    System.out.println(" memIdnum : " + memRow.getString("memIdnum"));
}
else if (memRow.getExtCode().equals("MemAttrRow"))
{
    System.out.print(" attrCode: " +
    memRow.getString("AttrCode"));
    System.out.println(" memSeqno: " + memRow.getInt("memSeqno"));
    if (memRow.getSegCode().equals("MemAttr"))
    {
        System.out.println(" attrVal: " +
        memRow.getString("AttrVal"));
    }
    else if (memRow.getSegCode().equals("MemName"))
    {
        System.out.print(" onmLast: " +
        memRow.getString("OnmLast"));
        System.out.println(" onmFirst: " +
        memRow.getString("OnmFirst"));
    }
}
// GetterException is thrown if the value can not be retrieved
        catch (GetterException ex)
Example: getting data from implementation-defined segments using web services

This code block gets data from the MemIdsWs array and displays the information contained for any IDS segment retrieved. The MemIdsWs Field Values are retrieved as Strings but Field Type can be used to determine the proper data type and cast as appropriate.

```java
//Get all the IDS rows and loop through them
MemIdsWs[] memIdsWs = members[i].getMemIds();
for (int k = 0; memIdsWs !=null && k < memIdsWs.length; k++)
{
    System.out.println("MemIdsWS.getAttrCode:" + memIdsWs[k].getAttrCode());
    //get all the Field Names for this IDS row in a string array.
    //this returns the FldNames unique to this IDS row (not inherited fields).
    String[] fldNames = memIdsWs[k].getFldNames();
    //get all the Field Values for this IDS row in a string array.
    String[] idsValues = memIdsWs[k].getValues();
    //get all the Field Types for this IDS row in a string array.
    String[] fldTypes = memIdsWs[k].getFldTypes();
    //for each Field Value, print out its name, type, and value.
    for (int j = 0; j < idsValues.length; j++)
    {
        System.out.print(" FldName:" + fldNames[j]);
        System.out.print(" FldType:" + fldTypes[j]);
        System.out.print(" FldValues:" + idsValues[j]);
        //All Field Values start as Strings but can be cast to the native type
        //Cast values based on Field Type and print the value.
        if (fldTypes[j].equalsIgnoreCase("Date"))
        {
            SimpleDateFormat sdf21 = new SimpleDateFormat("yyyy-MM-dd hh:mm:ss a");
            try
            {
                Date customDate = sdf21.parse(idsValues[j]);
                System.out.println(" Date.toString:" + customDate);
            }
            catch (ParseException ex)
            {
                System.out.println(" ** DateExcpt *" + idsValues[j]);
            }
        }
        else if (fldTypes[j].equalsIgnoreCase("Short"))
        {
            short customShort = Short.parseShort(idsValues[j]);
            System.out.println(" short.toString:" + customShort);
        }
        else if (fldTypes[j].equalsIgnoreCase("int"))
        {
            int customInt = Integer.parseInt(idsValues[j]);
            System.out.println(" int.toString:" + customInt);
        }
        else
        {
```

Chapter 2. Getting started with the SDK  47
Example: retrieving implementation-defined segments using Java

The mpi_segxfld table contains definitions of all field names and types that are contained in a segment. These definitions can be learned programatically by the using the Row.getSegDef() method and the SegDef.getFldDefByName() method. The following illustrates how to obtain this meta data for all the rows contained in a given MemRowList object.

```java
RowIterator rowIter = outMemRows.rows();

// The following will dump out all the meta data for
// the memRows contained in the outMemRows list.
while (rowIter.hasMoreRows())
{
   MemRow memRow = (MemRow) rowIter.nextRow();

   // for each memRow - get the Segment Definitions
   SegDef segDef = memRow.getSegDef();

   // for each SegDef - dump out the Field Definitions of each
   // field contained in each memRow
   System.out.print("SegDef SegCode=" + segDef.getSegCode());
   System.out.print(" ExtCode=" + segDef.getExtCode());
   System.out.print(" ObjCode=" + segDef.getObjCode());
   System.out.print(" hasDates=" + segDef.hasDATES());
   System.out.print(" hasStrings=" + segDef.hasSTRINGS());
   System.out.println(" fldDefCnt=" + segDef.getFldDefCnt());
   int iFldCnt = segDef.getFldDefCnt();
   for (int i=0; i < iFldCnt; i++)
   {
      segDef.getFldDefByNo(i).dump();
      FldDef fldDef = segDef.getFldDefByNo(i);
   }
}
```

Example: retrieving implementation-defined segments using web services

The mpi_segxfld table contains definitions of all field names and types that are contained in a segment. These definitions can be learned programatically by the using the getSegmentDef operation. This operation retrieves segment definitions and field definitions for use in determining the structure of IDS. This operation can also be used to determine the structure of fixed (non-IDS) segments.

```java
try
{
   service = new IdentityHubServiceLocator();
   port = service.getIdentityHub();

   //build the request object used in filtering the results
   SegmentDefGetRequest sdgr = new SegmentDefGetRequest();

   // Assumes a Member Custom (MEMCUST) segment is defined in the Hub.
   // Set Request SegCodeFilter option: a list of case sensitive SegCodes or all
   sdgr.setSegCodeFilter("MEMCUST");

   // Set request ObjCodeFilter, valid options: mem, dic, aud, or all
   sdgr.setObjCodeFilter("mem");
}
```
// perform the request and get back a list of SegDefs
SegDefWs[] segDefWs = port.getSegmentDef(sdgr);
return segDefWs;
}
catch(Exception e)
{
System.err.println(e.getMessage());
e.printStackTrace();
}

Printing the results from the getSegmentDef operation would show the following definition for our MemCust segment.

*** SegDef Start ****
SegDefWs.getExtCode:MemAttrRow getObjCode:mem getObjRecno:13573
getSegCode:MEMCUST getSegRecno:170
FldDefWs.getFldName:rowInd getFldType:int getFldOffset:0 getFldSize:0
FldDefWs.getFldName:entRecno getFldType:int getFldOffset:4 getFldSize:0
FldDefWs.getFldName:memRecno getFldType:int getFldOffset:8 getFldSize:0
FldDefWs.getFldName:memSeqno getFldType:int getFldOffset:12 getFldSize:0
FldDefWs.getFldName:cAudRecno getFldType:int getFldOffset:16
FldDefWs.getFldName:mAudRecno getFldType:int getFldOffset:20
FldDefWs.getFldName:srcFtime getFldType:Date getFldOffset:24
FldDefWs.getFldName:srcLtime getFldType:Date getFldOffset:32
FldDefWs.getFldName:recCtime getFldType:Date getFldOffset:40
FldDefWs.getFldName:recMtime getFldType:Date getFldOffset:48
FldDefWs.getFldName:recStat getFldType:String getFldOffset:56
FldDefWs.getFldName:attrRecno getFldType:int getFldOffset:58
FldDefWs.getFldName:asaIdxno getFldType:int getFldOffset:62 getFldSize:0
FldDefWs.getFldName:attrCode getFldType:String getFldOffset:66
FldDefWs.getFldName:custVal getFldType:String getFldOffset:79
FldDefWs.getFldName:custDate getFldType:Date getFldOffset:208
FldDefWs.getFldName:custScore getFldType:short getFldOffset:216
FldDefWs.getFldName:custRecno getFldType:int getFldOffset:218
*** SegDef End ****

---

**Web services Member class**

This is the class for the web services type member, which stores the information about a member. All of the operations related to a member make use of this class for input and output. The information about a member is in the form of member segments. The member class contains various methods to add and retrieve member segments as shown in the following code example.

```java
// Create a member object to hold member information.
Member member = new Member();
// Create MemHeadWs object to hold
// member header information.
MemHeadWs memHead = new MemHeadWs();

// Set the source code and member id in MemHeadWs object.
memHead.setSrcCode("RMC");
memHead.setMemIdnum("6315");
```
// Set the MemHeadWs object to the member.
member.setMemHead(memHead);

// Create MemNameWs object to hold member name.
MemNameWs[] memName = new MemNameWs[1];
memName[0] = new MemNameWs();

// Set the attribute code.
memName[0].setAttrCode("LGLNAME");

// Set the values for the first and last names.
memName[0].setOnmFirst("Rick");
memName[0].setOnmLast("Jones");

// Set the MemNameWs object to the Member.
member.setMemName(memName);

// Create MemDateWs object to hold member // date/time information.
MemDateWs[] memDate = new MemDateWs[1];
memDate[0] = new MemDateWs();

// Set the attribute code and date.
memDate[0].setAttrCode("BIRTHDT");
memDate[0].setDateVal("06/20/2001");

// Set the MemDateWs object to the Member.
member.setMemDate(memDate);

// Create MemAddrWs object to hold the member's address.
MemAddrWs[] memAddr = new MemAddrWs[1];
memAddr[0] = new MemAddrWs();
memAddr[0].setAttrCode("HOMEADDR");
memAddr[0].setStLine1("6401 W. Parkly ");
memAddr[0].setCity("Los Angeles ");
memAddr[0].setState("California ");
memAddr[0].setZipCode("95000");
member.setMemAddr(memAddr);

// Create MemIdentWs object to hold the member's id.
MemIdentWs[] memIdent = new MemIdentWs[1];
memIdent[0] = new MemIdentWs();
memIdent[0].setAttrCode("SSN");
memIdent[0].setIdIssuer("3");
memIdent[0].setIdNumber("501052342 ");
member.setMemIdent(memIdent);

// Create MemPhoneWs object to hold the member's // phone details.
MemPhoneWs[] memPhone = new MemPhoneWs[1];
memPhone[0] = new MemPhoneWs();
memPhone[0].setAttrCode("HOMEPHON");
memPhone[0].setPhArea("500");
memPhone[0].setPhNumber("222-2344");
memPhone[0].setPhExtn("x123");
member.setMemPhone(memPhone);

// Create MemAttrWs object to hold the member's // attributes.
MemAttrWs[] memAttr = new MemAttrWs[1];
memAttr[0] = new MemAttrWs();
memAttr[0].setAttrCode("SEX");
memAttr[0].setAttrVal("M");
member.setMemAttr(memAttr);
Web services Dictionary class

This is the class for the web services type dictionary. This class stores the dictionary information. All of the operations related to the dictionary make use of this class for input and output. The dictionary information is in the form of dictionary segments. The dictionary class has methods to add and retrieve dictionary segments as shown in the following code example.

```java
// Create Dictionary object which holds dictionary information.
Dictionary dictionary = new Dictionary();

// SrcHeadWs array which holds the header information
SrcHeadWs[] srcHead = new SrcHeadWs[2];

// Set various fields for the first SrcHeadWs segment.
srcHead[0] = new SrcHeadWs();
srcHead[0].setSrcRecno(255);
srcHead[0].setMaxLen(10);
srcHead[0].setMemTypeno(1);
srcHead[0].setMinLen(1);
srcHead[0].setPhyCode("TST255");
srcHead[0].setPoCmpCode(" ");
srcHead[0].setPoScore((short)0);
srcHead[0].setRiCheck("Y");
srcHead[0].setSrcCode("TST255");
srcHead[0].setSrcName("Test Mental Health 255");
srcHead[0].setSrcType("D");
srcHead[0].setRecStat("A");
srcHead[0].setRowInd("A");

// Set the type of operation indicating whether to Add or Delete a row
srcHead[0].setRowInd("A");

// Set various fields for the second SrcHeadWs segment
srcHead[1] = new SrcHeadWs();
srcHead[1].setSrcRecno(256);
srcHead[1].setMaxLen(10);
srcHead[1].setMemTypeno(1);
srcHead[1].setSrcCode("TST256");
srcHead[1].setPhyCode("TST256");
srcHead[1].setPoCmpCode(" ");
srcHead[1].setPoScore((short)0);
srcHead[1].setRiCheck("Y");
srcHead[1].setSrcName("Test Mental Health 256");
srcHead[1].setSrcType("D");
srcHead[1].setRecStat("A");
srcHead[1].setRowInd("A");

// Set the type of operation
srcHead[1].setRowInd("A");

// Add the source header information to the dictionary
dictionary.setSrcHead(srcHead);

// Create UsrHeadWs object which holds user header information
UsrHeadWs[] usrHead = new UsrHeadWs[1];
usrHead[0] = new UsrHeadWs();
usrHead[0].setUsrLogin("TSTUsr255");
usrHead[0].setUsrPass("TSTUsr255");
usrHead[0].setCvwPerm("A");
usrHead[0].setIxnPerm("A");
usrHead[0].setSegPerm("W");
usrHead[0].setIxnPerm("W");

// Set the type of operation
```
usrHead[0].setRowInd("A");

// Add the user header information to the dictionary
dictionary.setUsrHead(usrHead);

Web services request objects

All operations of the web services expect a request object as the input parameter. This request object passes on various parameters that a particular operation would require to execute successfully. All of the member information (in the form of a member object), and dictionary information (in the form of a dictionary object), along with various filters are added to this request object, which is then passed to the operation as an argument. The following example shows how to create a memberMatchRequest object, add various input parameters to it and pass it to the matchMember operation as an argument.

// Create a request object using which we pass input
// parameters to the operation.
MemberMatchRequest request = new MemberMatchRequest();

// Set the srccode and member id for the operation.
request.setSrcCode("RMC");
request.setMemIdnum("224422");

// Set a segment code filter to limit
// output to specific segments.
request.setSegCodeFilter("MEMHEAD, MEMADDR, MEMATTR, MEMDATE," + "MEMIDENT, MEMNAME, MEMPHONE, MEMXTSK");

// Set username & password for the operation.
request.setUserName("rwuser");
request.setUserPassword("rwuser");

// Set entity type as Identity (id)
// Entity types are listed in mpi_enttype table.
request.setEntType("id");

// Set the member type as PERSON.
// Member types are listed in mpi_memtype table.
request.setMemType("PERSON");

// Perform the matchMember operation.
return port.matchMember(request);

Quiz yourself: getting started

The following quizzes can help you determine your understanding of the basics of the IBM Initiate SDK.

SDK for Java overview quiz

1. How does the SDK for Java interact with the Master Data Engine?
   a. By connecting to the RDBMS
   b. Using RPC services on the Master Data Engine server
   c. By connecting to MPINET using a Context Object

2. A MemRowList can contain information about how many members?
   a. Zero
   b. More than one
   c. One
   d. All of the above
3. A segment maps to ___ of the Hub database.
   a. Group of tables
   b. A single table
   c. No such mapping is possible
4. Which segment contains information about the different segments in the Hub database?
   a. MemHead
   b. SegHead
5. How do you pass the information about a member to an interaction?
   a. Using the MemHead object
   b. Using the UsrHead object
   c. Using the MemRowList object

Answers:
1. ‘C’ – By connecting to MPINET using a Context Object
2. ‘D’ – All of the above
3. ‘B’ – A single table
4. ‘B’ - SegHead
5. ‘C’ – Using the MemRowList object

Web services overview quiz

1. Web services are built on which framework?
   a. Struts
   b. MVC
   c. Apache Axis
2. A client to web services
   a. Should use Apache Axis as SOAP engine
   b. Should be written in Java
   c. Can be any program which can generate SOAP messages according to web services WSDL file
3. A segment maps to ___ of the database
   a. Group of tables
   b. A single table
   c. No such mapping is possible
4. Which segment contains information about different segments in the database?
   a. MemHeadWs
   b. SegHeadWs
5. How do you pass the information about a member to an operation?
   a. Using MemHeadWs
   b. Using UsrHeadWs
   c. Using Member

Answers:
1. ‘C’ – Apache Axis
2. ‘C’ – Can be any program which can generate SOAP messages according to web services WSDL file
3. ‘B’ – A single table
4. ‘B’ - SegHeadWs
5. ‘C’ – Using Membert
Chapter 3. Basic interactions

Deploying IBM Initiate Master Data Service at the Get Well Hospital proved to be a great idea. Today doctors, surgeons and nurses feel it was an excellent step toward providing improved medical care for their patients. IBM Initiate Master Data Service successfully linked the profiles of the patients in the hospital. Now surgeons can search for any patient and get a 360-degree view of his or her encounters with the hospital.

To view the medical history of John Bach, they performed various interactions with the Master Data Engine, and retrieved the profile of John Bach and all related profiles. Based on the clear picture of John Bach’s medical history, the surgeons were able to perform a successful operation. After a day’s medical care in the Intensive Care Unit, John Bach was moved to the General Ward.

Meanwhile, John Bach’s newly constructed home was ready for occupation. His wife, son and daughter felt that John Bach would recover faster in the comfort of their own home. Therefore, once John Bach was moved to the General Ward they began moving their residence to the new home.

Per his physician’s advice, John Bach registered at the Nutrition department. He gave his full details, including his new address. The updated details of John Bach were submitted to the Master Data Engine from the Nutrition department source system.

Whenever the surgeons and physicians needed to know their patients’ medical histories, they approached the IT department personnel of Get Well Hospital, who retrieved the information using various interactions of the SDK. Now we’ll look into the basic interactions that you can perform in your custom applications using the SDK.

Note: This section covers features and benefits of the types of operations available on the SDK distribution disc. The code samples shown in this section are for demonstration only. Refer to the SDK distribution disc for compilable code samples.

Note: You cannot use hashed passwords to run an interaction.

To use an encrypted password in the ContextManager.props file, you must re-encrypt your plain-text password with the madpwd2 utility.

- Store a plain-text password in the server1.UsrPass=<password> property.
- Store a madpwd2-encrypted password in the server1.UsrPass2=<encrypted.password> property.

You can compile and run the examples that are packaged on the SDK distribution disc, which will help you understand how the interactions work. For Java users, refer to the ReadMe.Examples.txt file on the disc for instructions on configuring your system and compiling the examples. Web services SDK users should consult the IdentityHub.wsdl file on the disc.
Assumptions about the examples in this book

The examples provided on the SDK distribution CD are well-commented and more complete than those shown in this document. We provide examples in this document as a quick reference, to give you a general idea of how the interactions are used. They are not intended to be used as-is.

The following assumptions are made about these short examples:

- The necessary import or using statements are used but not shown. Refer to the BaseExamples.java or WsExampleBase.java file on the distribution CD for details on which statements are used.
- Some of the methods called in the examples were pre-written, including:
  - info(String msg)
  - err(String msg)
  - ixnError(String msg, String errorCode, String errorMessage)
- Some of the objects used were pre-declared and instantiated, including
  - context object (ctx)
  - context pool object (contextPool)
  - dictionary store object (dicStore)
- All segments used are fixed, such as MemName, MemAddr and UsrHead. The examples on disc also demonstrate how to create objects for implementation defined segments.

Member interactions

The following interactions pertain to manipulating member records. Most of the examples assume that the member is a person that has attributes such as birth date, sex, age, and so on.

The examples instantiate one or more MemHead objects, used to create new members. MemHead has three constructors:

- MemHead() – Creates a new MemHead object and sets a temporary memRecno. When the MemHead is inserted into the database, the temporary memRecno is converted to the next available number. This one is most commonly used in the examples.
- MemHead (long memRecno) – Creates a new MemHead object and sets the specified memRecno. (If this value is zero (0), then a reserved memRecno is generated. When the MemHead is inserted into the database, the memRecno is converted to the next available number.)
- MemHead (long memRecno, long entRecno) – Creates a new MemHead object and sets the specified memRecno and entRecno. (If the memRecno or entRecno value is zero (0), then a reserved memRecno or entRecno will be generated. When the MemHead is inserted into the database, the memRecno or entRecno is converted to the next available number.)

MemName and MemDate objects are frequently used as well.

MemName has four constructors, but only one is used in the examples. Please consult the Javadoc for information on the other constructors.

- public MemName(MemHead memHead) – Constructs a MemName object and sets the member identification info. The MemRecno and EntRecno information is set using the MemHead object.
MemDate has four constructors, but only one is used in the examples. Please consult the Javadoc HTML documentation for information on the other constructors.

- public MemDate(MemHead memHead) – Constructs a MemDate object and sets the member identification info. The MemRecno and EntRecno information is set using the MemHead object.

**Note:** When storing dates in the Master Data Engine, use the normalized format YYYY-MM-DD. If dates are stored in other formats, they may not be caught by dateval validation and will not be displayed properly in client applications, such as Inspector.

This example code provided with the SDK:

```java
MemAttrRow memDate = getDicStore().createMemAttrRowByName("Birth Date", memHead);
memDate.setString("dateVal", "17-08-1973");
```

should instead read:

```java
MemDate memDate = new MemDate();
memDate.setDateVal("dateVal", "1973-08-17");
```

Refer to the "mpi_memdate and mpi_memdate_h" topic within the Data Model Description, and the madison.mpi.MemDate class in the Javadoc Information for details.

**Member search**

To get a 360-degree view of John Bach, the surgeons approached the IT department at Get Well Hospital, who first made a preliminary search for John Bach’s profile using a member search interaction by specifying some basic criteria. This search resulted in a list of members that matched the criteria.

You can use the member search interaction object IxnMemSearch to return a list of member rows that fit a list of search criteria. For instance, in the following example we show a legal name search for another member whose name is Norman.

**Note:** A minimum score (minscore) setting of 0 previously meant that all scores were considered during a member search interaction. Thus, all potential candidates were returned in the search regardless of score. This behavior lead to unnecessary I/O, which had a negative impact on performance. The Engine logic has been modified such that a minscore of 0 now returns only members that score at or above 0. If a user has the need to return all candidates, a large negative score such as -999 should be used.

**memSearch Java example**

```java
// Create the member search interaction object.
IxnMemSearch memSearch = new IxnMemSearch(ctx);

// Create member rowlists to hold input and output row(s).
MemRowList inpMemRows = new MemRowList();
MemRowList outMemRows = new MemRowList();
MemHead memHead = new MemHead();

// You should get in the habit of setting to 0 the recno you are not searching on.
MemHead memhead = new MemHead();
memHead.setMemRecno(0);
memHead.setEntRecno(0);
```
// Set entity type as Identity (id).
// Entity types are listed in mpi_enttype table.
memSearch.setEntType("id");

// Set the member type as PERSON.
// Member types are listed in mpi_memtype table.
memSearch.setMemType("PERSON");

// Set the SegCodeFilter to the segment codes
// we want to display in the output.
// In the result set we get, we want only fields
// from member head table and name table.
memSearch.setSegCodeFilter("MEMHEAD,MEMNAME");

// Set the record status indicators desired.
// The values include (A)ctive, (I)nactive, (D)eleted and (S)hadow.
memSearch.setRecStatFilter("A,I");

// Set memstat value as A - Active, O - Overlay.
memSearch.setMemStatFilter("A,O");
memSearch.setMinScore((short) 0);
memSearch.setMaxRows(50);

// Searches require well-formed member objects
// so there must be a memhead even if it is empty.
inpMemRows.addRow(memHead);

// The search is performed using the member's legal name,
// SSN and birth date.
// Construct/initialize a memName record and add
// it to the inpMemRow list.
MemName memName = new MemName(memHead);
memName.setAttrCode("LGLNAME");
memName.setOnmLast("Beverly");
memName.setOnmFirst("Norman");
inpMemRows.addRow(memName);

// Construct/initialize a memIdent record
// and add it to the inpMemRow list.
MemIdent memIdent = new MemIdent(memHead);
memIdent.setAttrCode("SSN");
memIdent.setIdIssuer("SSA");
memIdent.setIdNumber("359426486");
inpMemRows.addRow(memIdent);

// Construct/initialize a memDate record and
// add it to the inpMemRow list.
MemDate memDate = new MemDate(memHead);
memDate.setAttrCode("BIRTHDT");
memDate.setDateVal("1962-11-04");
inpMemRows.addRow(memDate);

// Execute member search interaction with the options
// GetType.ASENTITY - specifies that if any member of
// an Entity meets the requirements of the interaction, all
// related members in the Entity will also be retrieved
// SearchType.ASMEMBER- tells the search to compare
// the input on a member by member basis.
memSearch.execute(inpMemRows, outMemRows,
    GetType.ASENTITY, SearchType.ASMEMBER);

The resulting rows will be stored in outMemRows upon successful completion of
the interaction.

Possible values for GetType are GetType.ASMEMBER and GetType.ASENTITY. 
GetType.ASMEMBER will return only members that match your search criteria.
GetType.ASENTITY will return members that match the search criteria and any members that are linked to those matching members. Currently the only valid Search Type available is ASMEMBER.

**searchMember web services example**

**Note:** Example code for this operation is also available in C# (Microsoft .NET). Refer to the following file for the complete code: “exMemSearch.cs.”

```csharp
// Create request object using which we pass input
// parameters to the operation.
MemberSearchRequest request = new MemberSearchRequest();

// Create Member object for the Operation.
Member member = new Member();

// Set the various fields for the MemHeadWs.
MemHeadWs memHead = new MemHeadWs();
memHead.setMemRecno(0);
memHead.setEntRecno(0);

// Set the MemberHeadWs object to the member.
member.setMemHead(memHead);

// The search is performed based on the member's
// legal name and birth date.
// Construct/initialize a memName record and set
// it to the member.
MemNameWs[] memName = new MemNameWs[1];
memName[0] = new MemNameWs();
memName[0].setAttrCode("LGLNAME");
memName[0].setOnmFirst("Norman");
memName[0].setOnmLast("Beverly");
member.setMemName(memName);

// Construct/initialize a memDate record and
// set it to the member.
MemDateWs[] memDate = new MemDateWs[1];
memDate[0] = new MemDateWs();
memDate[0].setAttrCode("BIRTHDT");
memDate[0].setDateVal("1962-11-04");
member.setMemDate(memDate);

MemIdentWs[] memIdent = new MemIdentWs[1];
memIdent[0] = new MemIdentWs();
memIdent[0].setAttrCode("SSN");
memIdent[0].setIdNumber("359426486");
memIdent[0].setIdIssuer("SSA");
member.setMemIdent(memIdent);

// Set the Member object to the request.
request.setMember(member);

// Set a segment code filter to limit output
// to specific segments.
request.setSegCodeFilter("MEMHEAD,MEMADDR,MEMATTR,MEMDATE,MEMIDENT," + "MEMNAME,MEMPHONE,MEMXTSK");

// Set the username & password for the operation
request.setUserName("rwuser");
request.setUserPassword("rwuser");

// Set entity type as Identity (id).
// Entity types are listed in mpi_enttype table.
request.setEntType("id"); //hh or id.

// Set the member type as PERSON.
// Member types are listed in mpi_memtype table.
```
request.setMemType("PERSON");

// Specify that if any member of an Entity meets
// the requirements of the operation, all related members
// in the Entity should also be retrieved.
request.setGetType("ASENTITY");

// Set the minimum score to limit output.
request.setMinScore(new Short((short) 1));

// Perform the searchMember operation.
port.searchMember(request);

Possible values for GetType are GetType.ASMEMBER and GetType.ASENTITY.
GetType.ASMEMBER will return only members that match your search criteria.
GetType.ASENTITY will return members that match the search criteria and any
members that are linked to those matching members.

Note: Refer to “Web services basics” on page 25.

**Member get**

After the IT personnel searched for John Bach's records through the member search
interaction, they provided the surgeons with John Bach's source ID and member ID
and asked them to refer to the source ID and member ID when they needed John
Bach's medical records. The IT department used the member get interaction to
retrieve John Bach's records.

You can use the member get interaction object IxnMemGet to retrieve individual
members and their attributes based on a known member value. The difference
between a member search interaction and a member get interaction is the member
get interaction retrieves data using a member key value, such as an EntRecno, a
SrcCode and MemIdnum, or a MemRecno. In contrast, the member search
interaction returns the members who—in comparison with the input data—have a
score higher than the minimum score defined.

**memGet Java example**

// Create a member get interaction object.
IxnMemGet memGet = new IxnMemGet(ctx);

// Create member rowlists to hold input
// and output row(s).
MemRowList inpMemRows = new MemRowList();
MemRowList outMemRows = new MemRowList();

// Create a MemHead object.
// MemHead models the Initiate database table mpi_memhead.
MemHead memHead = new MemHead();
memHead.setSrcCode("SURG");
memHead.setMemIdnum("144760");
memHead.setEntRecno(0);
inpMemRows.addRow(memHead);

// Set a segment code filter to limit
// output to specific segments.
memGet.setSegCodeFilter("MEMHEAD,MEMATTR,MEMNAME,MEMADDR,MEMPHONE,"
+ "MEMIDENT,MEMDATE");

// Set the record status indicators desired.
// The values include (A)ctive, (I)nactive, (D)eleted and (S)hadow.
memGet.setRecStatFilter("A");
memGet.setMemType("PERSON");
// Set entity type as Identity (id).
// Entity types are listed in mpi_enttype table.
memGet.setEntType("id");

// GetType.ASMEMBER - ASMEMBER specifies that only individual
// members will be retrieved.
// KeyType.MEMIDNUM - specifies the srcCode/memIdnum is the
// retrieval key.
// Execute the member get interaction.
memGet.execute(inpMemRows, outMemRows, GetType.ASMEMBER,
    KeyType.MEMIDNUM);

The resulting rows will be stored in outMemRows upon successful completion of
the interaction.

Task segments can also be returned during a Member Get interaction by adding
the segment “MEMXTSK” to the setSegCodeFilter() methods list.

Remember that in order to get a segment in the results, you must add them to the
SegCode filter, for example,

memGet.setSegCodeFilter("MEMHEAD,MEMNAME,MEMXTSK");

In addition to listing specific segments you wish returned, you can now ask for all
of a particular type of segment. For best performance, it is preferable to explicitly
list the exact segments your application needs. However, for situations when you
can not know the segments in advance, you can now make a request that allows
you to generically get all of a specific type of segment.

MEMALL can be used for returning all member segments which are subclasses of
madision.mpi.MemRow. This option returns segments that most applications will
not use and you should check the results as it may return more rows then are
necessary.

MEMATTRALL can be used to return all member attribute segments. These
segments are those that are subclasses of madison.mpi.MemAttrRow. This option
returns the segments that most typical applications will use and represents a nice
balance between MEMALL and specifically listing segments, for example:

memGet.setSegCodeFilter("MEMALL");
memGet.setSegCodeFilter("MEMHEAD,MEMATTRALL");

getMember web services example

Note: Example code for this operation is also available in C# (Microsoft .NET).
Refer to the following files for the complete code: “exGetMember.cs.”

// Create a request object using which we pass input
// parameters to the operation.
MemberGetRequest request = new MemberGetRequest();

// Set the source code and member Id
// for the operation.
request.setSrcCode(sSrcCode);
request.setMemIdnum(sMemIdnum);

// Set a segment code filter to limit
// output to specific segments.
request.setSegCodeFilter(sSegCodeFilter);

// Set the username & password for the operation.
request.setUserName("rwuser");
request.setUserPassword("rwuser");
// Set entity type as Identity (id).
// Entity types are listed in mpi_enttype table.
request.setEntType("id"); //hh or id.

// Set the member type as PERSON.
// Member types are listed in mpi_memtype table.
request.setMemType("PERSON");

// Set the audit mode to none which means that the
// Creation/modification dates/times will be filled in.
request.setAudMode("AUDNONE");

Member[] members = null;

// Perform the getMember operation.
port.getMember(request);

Task segments can also be returned during a getMember operation by adding the
segment “MEMXTSK” to the setSegCodeFilter() methods list.

Remember that in order to get a segment in the results, you must add them to the
SegCode filter. The following code illustrates how to specify the segments.

For example:
request.setSegCodeFilter("MEMHEAD,MEMADDR,MEMATTR,MEMDATE,MEMIDENT,MEMNAME,
MEMPHONE,MEMXTSK");

A new option for SegCodeFilters has been introduced in version 7.5. In addition to
listing specific segments you wish returned, you can now ask for all of a particular
type of segment. For best performance, it is preferable to explicitly list the exact
segments your application needs. However, for situations when you can not know
the segments in advance, you can now make a request that allows you to
generically get all of a specific type of segment.

MEMALL can be used for returning all member segments which are subclasses of
madision.mpi.MemRow. This option will return segments that most applications
will not use and you should check the results as it may return more rows then are
necessary.

MEMATTRALL can be used to return all member attribute segments. These
segments are those that are subclasses of madison.mpi.MemAttrRow. This option
will return the segments that most typical applications will use and represents a
nice balance between MEMALL and specifically listing segments.

For example:
memGet.setSegCodeFilter("MEMALL");
memGet.setSegCodeFilter("MEMHEAD,MEMATTRALL");

**Member match**

A few days before the surgery, John Bach’s surgeons wanted to view all records
related to John Bach from all medical departments, but the only identifiers
available were the source ID and member ID of John Bach. The IT department
used the match member interaction to retrieve John Bach’s record using the source
ID and member ID. The engine automatically used the attributes of this member as
the input to retrieve all other related records.
You can use the member match interaction object IxnMemMatch to perform a two-phase search for members. This interaction first retrieves the input member using a key value like the member get interaction. The Master Data Engine automatically uses all the resulting member’s attributes to perform a search for other members with matching attributes. The Search portion of a MemMatch interaction does not pay attention to the maxRows or minScore API settings. It operates as the entity manager does—using the thresholds—and only returning those members that are above the CR threshold, or that take part in a Review Identifier task.

**memMatch Java example**

```java
// Create the member match interaction object.
IxnMemMatch memMatch = new IxnMemMatch(ctx);

// Create member rowlists to hold input
// and output row(s).
MemRowList inpMemRows = new MemRowList();
MemRowList outMemRows = new MemRowList();

// Create a MemHead object to pass in the member's ID numbers
// for whom we wish to find the matches.
// MemHead models Initiate database table mpi_memhead.
MemHead memHead = new MemHead();

// Set the identifiers of member to be matched.
memHead.setSrcCode("RMC");
memHead.setMemIdnum("112917");
memHead.setMemRecno(0);
memHead.setEntRecno(0);
inpMemRows.addRow(memHead);

// Set a segment code filter to limit
// output to specific segments.
memMatch.setSegCodeFilter("MEMHEAD,MEMATTR,MEMNAME," + 
"MEMADDR,MEMPHONE,MEMIDENT,MEMDATE");

// Set the record status indicators desired.
// The values include (A)ctive, (I)nactive, (D)eleted and (S)hadow.
memMatch.setRecStatFilter("A,I");
memMatch.setMinScore((short) 1);
memMatch.setMemType("PERSON");

// Set entity type as Identity (id)
// Entity types are listed in mpi_enttype table.
memMatch.setEntType("id");

// GetType.ASMEMBER- specifies that only individual
// members will be retrieved
// SearchType.ASMEMBER- tells the search to compare
// the input on a member by member basis..
// KeyType.MEMIDNUM - specifies the srcCode/memIdnum is the
// retrieval key
// Execute the member match interaction.
memMatch.execute(inpMemRows, outMemRows, GetType.ASMEMBER,
SearchType.ASMEMBER, KeyType.MEMIDNUM);
```

The resulting rows will be stored in outMemRows upon successful completion of the interaction. Additional Member Match information includes:

- Possible values for GetType are GetType.ASMEMBER and GetType.ASENTITY:
  - **GetType.ASMEMBER** – Returns only members that match the search criteria
  - **GetType.ASENTITY** – Returns members that match the search criteria and any members that are linked to those matching members
- Currently the only valid Search Type available is ASMEMBER.
The key type defines which key value is to be used for the search and can be one of the following:
- **KeyType.MEMRECNO** – MemRecno is the retrieval key
- **KeyType.ENTRECNO** – EntRecno is the retrieval key
- **KeyType.MEMIDNUM** – SrcCode and MemIdnum is the retrieval key
- **KeyType.UNKNOWN** – Placeholder value which shows that KeyType is not yet set

**matchMember web services example**

```java
// Create a request object using which we pass input parameters to the operation.
MemberMatchRequest request = new MemberMatchRequest();

// Set the srccode and member id for the operation.
request.setSrcCode("RMC");
request.setMemIdnum("112917");

// Set a segment code filter to limit output to specific segments.
request.setSegCodeFilter("MEMHEAD,MEMADDR,MEMATTR,MEMDATE," + "MEMIDENT,MEMNAME, MEMPHONE,MEMXTSK");

// Set the username & password for the operation.
request.setUserName("rwuser");
request.setUserPassword("rwuser");

// Set entity type as Identity (id).
// Entity types are listed in mpi_enttype table.
request.setEntType("id");

// Set the member type as PERSON.
// Member types are listed in mpi_memtype table.
request.setMemType("PERSON");

// Perform the matchMember operation.
port.matchMember(request);
```

**Comparing member interactions**

The following table provides a comparison of member get, search and match interactions.

<table>
<thead>
<tr>
<th>Interaction</th>
<th>Purpose</th>
<th>Ixn</th>
<th>Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>Member Search</td>
<td>To return a list of member rows that fit a list of search criteria</td>
<td>IxnMemSearch</td>
<td>Generic search criteria like: minimum score, member name, member address, member phone</td>
</tr>
<tr>
<td>Member Get</td>
<td>Retrieve individual members and their attributes</td>
<td>IxnMemGet</td>
<td>Known member values like MemRecno, or SrcCode and MemIdNum</td>
</tr>
<tr>
<td>Member Match</td>
<td>Performs a member get for a single member then uses all the attributes from the get to perform a search for other like members</td>
<td>IxnMemMatch</td>
<td>Known member values like MemRecno, or SrcCode and MemIdNum</td>
</tr>
</tbody>
</table>
Member put

When John Bach registered at the Nutrition department, the updated details with new address had to be submitted to the Hub database. The IT department at Get Well Hospital used the member put interaction to submit the modified member information.

You can use the IxnMemPut interaction object to submit modified member rows to the Master Data Engine.

Note: Refer to Appendix B, “Member put interaction—options and behavior,” on page 129 for detailed Member Put behavior and different put-mode types.

For current examples, refer to the SDK distribution disc.

Qualified member put

To help prevent redundant and duplicate records, the Qualified member put interaction (IxnMemPutQual) was developed. IxnMemPutQual provides a process for inserting and updating member data, without uniquely identifying the member by using a memRecno or a srcCode+memIdnum combination. This interaction can evaluate one incoming record at a time and, by using an integrated search feature, determine whether adding it will create a redundancy in the Hub.

The IxnMemPutQual interaction takes the supplied member attributes and performs a search in the Hub, looking for the most likely candidate to update. PutType enables flexibility in selecting the record to be updated. The candidate search can result in one of three scenarios:

- No candidate found. In this case a new member will be added to the Hub using the SrcCode and a generated MemIdNum. Again, if the MemIdNum is supplied in the input, an EINVAL error is returned and the record is not added. PutType enables flexibility in selecting the record to be updated.
- One candidate found (the highest scoring candidate). In this case one MemHead is returned and the member is updated with the new attribute values based on defined Engine business rules.
- More than one candidate found (two or more candidates share the highest score). In this case there are two outcomes that depend upon the put type:
  - If the put type is UNCLEAR_WINNER, an EEXISTS error is returned along with the candidates' MemHead objects (multiple MemHeads). No member is updated or inserted into the database.
  - If the put type is DEFAULT_WINNER, then the candidate with the lowest MemRecno is updated with the new attribute values.

MemPutQual honors all implicit member modes (memMode); ATTRCOMP, PARTIAL, and COMPLETE. Audit history works the same as it does for IxnMemPut.

For current examples, refer to the SDK distribution disc.

Member unput

When John Doe registered at Get Well Hospital’s Cardiology department, his LGLNAME was changed to Johnny Doe. When the IT department realized that Mr. Doe’s name was changed in error, they used the member unput interaction to reset
the LGLNAME back to John Doe. In doing this, the LGLNAME attribute value of Johnny Doe was logically deleted (its record status is ‘D’) and the value of John Doe was set to Active (its record status is ‘A’).

You can use the IxnMemUnput interaction object to perform a MEMUNPUT (Member unput) by passing in a valid audRecno. History must be enabled to use MemUnput for updates, but history is not required to use MemUnputs for inserts. All attribute changes or inserts for a member are also backed out when the member record change or insert is backed-out via MemUnput.

Note: Potential Overlays cannot be backed out using MemUnput or MemUndo.

- For a newly inserted record, the IxnMemUnput interaction will drop the member record. IxnMemUnput can undo an insert for MEMPUT, MEMPUTBULK or MEMPUTQUAL interactions. The Audlevel for MEMPUT, MEMPUTBULK or MEMPUTQUAL should be set to 2 or higher in mpi_ixnhead. If MemUnput is called following a MEMPUTBULK, we attempt to back out all records affected by the MEMPUTBULK call. The records that can be backed out will be. If backing out changes to one of the member records fails, we cannot roll back the unput of the MEMPUTBULK transaction.
- For a member update, the IxnMemUnput interaction will backout changes to all attributes and revert the member to the previous state.
- To undo an update to an attribute, history must be turned on for both the member type and the attribute.
- To undo an update to a member record, the audRecno must be associated with the last update to that record.
- The interaction would fail if attempts are made to undo an update to an attribute with nsactive (in the mpi_segattr table) not set to 1.

This interaction is available for the SDKs for Java only. For current examples, refer to the SDK distribution disc.

**Member compare**

In the Dental department of Get Well Hospital there were two records with the same patient name as J Bach. One of them belongs to John Bach and another to Jonathan Bach. To differentiate John Bach’s record from that of Jonathan Bach, the IT department used the member compare interaction.

You can use the IxnMemCompare interaction object to retrieve a text representation of the Master Data Engine results of comparing two members. Direct comparisons of attributes for member one (proband) are compared against those of member two (candidate) to determine if the two records represent the same person.

**memComp Java example**

```java
// Create a member compare interaction object.
IxnMemCompare memComp = new IxnMemCompare(ctx);

// Create member rowlists to hold input
// and output row(s).
MemRowList inpMemRows = new MemRowList();
MemRowList outMemRows = new MemRowList();

// Set the member type as PERSON.
// Member types are listed in mpi_memtype table.
memComp.setMemType("PERSON");
```
// Set entity type as Identity (id).
// Entity types are listed in mpi_enttype table.
memComp.setEntType("id");

// Create two member objects to compare. The following
// two members are very much identical. So we get a high
// score.
// This members' attributes are Source No: 214
// EID: 1000000005, FirstName: Vidalia
// LastName: Garay-Deras, MiddleName: G.
MemHead memHead = new MemHead();
memHead.setSrcCode("FGH");
memHead.setMemIdnum("988608");
memHead.setEntRecno(0);
inpMemRows.addRow(memHead);

// This members' attributes are Source No: 217
// EID: 1000000005, FirstName: Vidalia
// LastName: Garay-Deras, MiddleName: D.
MemHead memHead2 = new MemHead();
memHead2.setSrcCode("PHYS");
memHead2.setMemIdnum("145212");
inpMemRows.addRow(memHead2);

// This would execute the comparison on the input
// members and the output is stored in outMemRows.
if (memComp.execute(inpMemRows, outMemRows, KeyType.MEMIDNUM))
{

    // Print some internal details
    // of the comparison process.
    System.out.println("CompareInfo: " + memComp.getCompareInfo());
    // Print the output rows.

    RowIterator rowIter = outMemRows.rows();
    while (rowIter.hasMoreRows())
    {
        MemRow memRow = (MemRow) rowIter.nextRow();

        if (memRow instanceof MemHead)
        {
            MemHead memHead3 = (MemHead) memRow;

            // The match score is located in the
            // MEMHEAD object returned in the output rowlist.
            System.out.print(" memRecno: " + memHead3.getMemRecno());
            System.out.print(" memSeqno: " + memHead3.getMemSeqno());
            System.out.print(" srcCode: " + memHead3.getSrcCode());
            System.out.print(" memIdnum: " + memHead3.getMemIdnum());
            System.out.print(" matchScore: " + memHead3.getMatchScore());
        }
        System.out.println();
    }
}

Note: System.out.println() statements can be output in the mpinet.out file. Any
System.out.print() executed in a handler will be output in mpinet.out.

The resulting rows will be stored in outMemRows upon successful completion of
the interaction. The following shows the output:

**compareMember web services example**

// Create a request object using which we pass input
// parameters to the operation.
MemberCompareRequest request = new MemberCompareRequest();
// Create two member objects to compare. The following two
// members are very much identical. So we get a high score.
// This member's attributes are Source No: 214
// EID: 1000000005, FirstName: Vidalia
// LastName: Garay-Deras, MiddleName: G.
request.setSrcCodeA("FGH");
request.setMemIdnumA("988608");

// This member's attributes are Source No: 217
// EID: 1000000005, FirstName: Vidalia
// LastName: Garay-Deras, MiddleName: D.
request.setSrcCodeB("PHYS");
request.setMemIdnumB("145212");

// Set the username & password for the operation.
request.setUserName("rwuser");
request.setUserPassword("rwuser");

// Set entity type as Identity (id).
// Entity types are listed in mpi_enttype table.
request.setEntType("id"); // hh or id.

// Set the member type as PERSON.
// Member types are listed in mpi_memtype table.
request.setMemType("PERSON");

// Perform the compareMember operation
port.compareMember(request);

The following describes the SOAP message format.

Sequence of operations

The Axis library creates and sends a request with the input parameters to the
server to perform an operation. (A user can also send an XML request directly to
web services using the SOAP message format given below.) The response message
is returned to the client after the operation is performed. Both the request and
response messages are in XML format.

The general structure of a SOAP message is as follows:

```xml
<SOAPENV:Envelope Attributes
    <SOAPENV:Body Attributes
    </SOAPENV:Body>
</SOAPENV:Envelope>
```

SOAP Header element

The SOAP Header element contains application-specific information (such as
authentication and application name) about the SOAP message. This part of the
message is optional.

SOAP Envelope element

The SOAP Envelope element is the root element of a SOAP message. It defines the
XML document as a SOAP message. This part of the message is mandatory.

SOAP Body element

The SOAP Body element contains the actual SOAP message intended for the
ultimate endpoint of the message. (This part of the message is mandatory.)
SOAP request message

Following is the SOAP request message (HTTP message+SOAP message), which is used to perform the compareMember operation.

- The compareMember tag is the operation name that is being requested
- The contents of compareMember structure are serialized according to the SOAP encoding rules
- The xmlns attribute contains the following namespace uniform resource indicator (URI): “urn:bean.initiate.com”
- The children of memCompReq tag contains various input parameters to the operation

Following is the list of tags contained in the memCompReq tag; each tag contains information relevant to the members being compared.

<table>
<thead>
<tr>
<th>Tag</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>username</td>
<td>User information</td>
<td>“rwuser”</td>
</tr>
<tr>
<td>userPassword</td>
<td>Password information</td>
<td>“rwuser”</td>
</tr>
<tr>
<td>srcCodeA</td>
<td>First member’s source code</td>
<td>“FGH”</td>
</tr>
<tr>
<td>srcCodeB</td>
<td>Second member’s source code</td>
<td>“PHYS”</td>
</tr>
<tr>
<td>entType</td>
<td>Entity information</td>
<td>“id”</td>
</tr>
<tr>
<td>memType</td>
<td>Member type</td>
<td>“PERSON”</td>
</tr>
<tr>
<td>memIdnumA</td>
<td>First member’s member ID</td>
<td>“988608”</td>
</tr>
<tr>
<td>memIdnumB</td>
<td>Second member’s Member ID</td>
<td>“145212”</td>
</tr>
</tbody>
</table>

SOAP message embedded in HTTP Request

```xml
<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope
    xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <soapenv:Body>
        <compareMember xmlns="urn:bean.initiate.com">
            <memCompReq xmlns="">
                <userPassword>rwuser</userPassword>
                <username>rwuser</username>
                <audMode xsi:nil="true"/>
                <srcCodeA>FGH</srcCodeA>
                <memRecnoB xsi:nil="true"/>
                <memIdnumA>988608</memIdnumA>
                <entType>id</entType>
                <memType>PERSON</memType>
                <srcCodeB>PHYS</srcCodeB>
                <memRecnoA xsi:nil="true"/>
                <memIdnumB>145212</memIdnumB>
            </memCompReq>
        </compareMember>
    </soapenv:Body>
</soapenv:Envelope>
```

SOAP response message

Following is the SOAP response message, containing the output from the compareMember operation.
- The compareMemberResponse tag defines the compareMember response.
- The contents of compareMember structure are serialized according to the SOAP encoding rules.
- The xmlns:ns1 attribute contains the following namespace uniform resource indicator (URI): “urn:bean.initiate.com”
- The compareMemberReturn tag contains a list of item tags each corresponding to the members compared.
- Each item tag contains the comparison information along with the member header information.
- The compareInfo tag contains the string, which has the information about the comparison results like the score.

Following is the list of tags contained in the first memHead tag; each tag contains information relevant to the compared members.

<table>
<thead>
<tr>
<th>Tag</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>caudRecno</td>
<td>Current® audit record number</td>
<td>137</td>
</tr>
<tr>
<td>matchScore</td>
<td>Match score</td>
<td>221</td>
</tr>
<tr>
<td>maudRecno</td>
<td>Modified audit record number</td>
<td>151</td>
</tr>
<tr>
<td>memIdnum</td>
<td>Member ID number</td>
<td>988608</td>
</tr>
<tr>
<td>memRecno</td>
<td>Member record number</td>
<td>18</td>
</tr>
<tr>
<td>memSeqno</td>
<td>Member sequence number</td>
<td>29</td>
</tr>
<tr>
<td>memStat</td>
<td>Member status</td>
<td>A</td>
</tr>
<tr>
<td>MemVerno</td>
<td>Version number for member</td>
<td>0</td>
</tr>
<tr>
<td>rowInd</td>
<td>Row indicator</td>
<td>I</td>
</tr>
<tr>
<td>srcCode</td>
<td>Source code</td>
<td></td>
</tr>
<tr>
<td>srcRecno</td>
<td>Source Record Number</td>
<td>214</td>
</tr>
</tbody>
</table>

SOAP embedded in HTTP Response

```xml
<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soapenv:Body>
    <compareMemberResponse xmlns="urn:bean.initiate.com">
      <compareMemberReturn xmlns=""/>
      <item xsi:type="ns1:Member" xmlns:ns1="urn:bean.initiate.com">
        <memHead>
          <caudRecno>137</caudRecno>
          <memIdnum>988608</memIdnum>
          <linkType></linkType>
          <entRecno>-294966362</entRecno>
          <matchCode>Unknown</matchCode>
          <maudRecno>137</maudRecno>
          <srcCode></srcCode>
          <rowInd>I</rowInd>
          <memVerno>0</memVerno>
          <srcRecno>214</srcRecno>
          <memSeqno>29</memSeqno>
          <memStat>A</memStat>
          <memRecno>18</memRecno>
          <matchScore>221</matchScore>
        </memHead>
      </item>
    </compareMemberResponse>
  </soapenv:Body>
</soapenv:Envelope>
```
Comparison info

The resulting rows will be stored in outMemRows upon successful completion of the interaction. The following shows the output:

<table>
<thead>
<tr>
<th>CompareInfo:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Comparing proband 18 with candidate 17, score=204.</td>
<td></td>
</tr>
<tr>
<td>SEX:</td>
<td>F</td>
</tr>
<tr>
<td>wgt= +1.00, mcc=E</td>
<td></td>
</tr>
<tr>
<td>PXNM:</td>
<td>GARAY</td>
</tr>
<tr>
<td>wgt=+12.01, mcc=D</td>
<td></td>
</tr>
<tr>
<td>ZIP:</td>
<td></td>
</tr>
<tr>
<td>wgt= +0.00, mcc=M</td>
<td></td>
</tr>
<tr>
<td>PHONE:</td>
<td>5355432</td>
</tr>
<tr>
<td>wgt= -1.09, mcc=D</td>
<td></td>
</tr>
<tr>
<td>SSN:</td>
<td>463827154</td>
</tr>
<tr>
<td>wgt= +6.05, mcc=E</td>
<td></td>
</tr>
<tr>
<td>DOB:</td>
<td>19610428</td>
</tr>
<tr>
<td>wgt= +4.08, mcc=E</td>
<td></td>
</tr>
<tr>
<td>ADDR:</td>
<td>933 HOLLY</td>
</tr>
<tr>
<td>wgt= -1.60, mcc=D</td>
<td></td>
</tr>
<tr>
<td>PXNM Detail:</td>
<td></td>
</tr>
<tr>
<td>PXNM[1]:</td>
<td>GARAY</td>
</tr>
<tr>
<td>wgt= +4.77, mcc=X</td>
<td></td>
</tr>
<tr>
<td>PXNM[2]:</td>
<td>DERAS</td>
</tr>
<tr>
<td>wgt= +4.77, mcc=X</td>
<td></td>
</tr>
<tr>
<td>PXNM[3]:</td>
<td>VIDALIA</td>
</tr>
<tr>
<td>wgt= +4.27, mcc=X</td>
<td></td>
</tr>
<tr>
<td>PXNM[4]:</td>
<td>G</td>
</tr>
<tr>
<td>D</td>
<td>wgt= -1.80, mcc=D</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>memRecno: 18</th>
<th>memSeqno: 29</th>
<th>srcCode:</th>
<th>memIdnum: 988608</th>
<th>matchScore: 204</th>
</tr>
</thead>
<tbody>
<tr>
<td>memRecno: 17</td>
<td>memSeqno: 37</td>
<td>srcCode:</td>
<td>memIdnum: 145212</td>
<td>matchScore: 204</td>
</tr>
</tbody>
</table>
Member merge

Several months before the surgery, John Bach made a few visits to the Get Well Hospital's Regional Medical Center (RMC). On his second visit, John Bach was asked to register again because he did not remember his registration number. Thus two sets of records for John Bach were created, each with a different Member ID.

To overcome this duplication or redundancy, you can use the IxnMemMerge interaction object to merge one member with another, which results in a surviving member and an obsolete member. The input RowList contains two MemHead segments. The first is the surviving member, and the second is the obsolete member.

Note: There is an implied keyType of MEMIDNUM on this interaction. A SrcCode and MemIdnum must be specified in order to merge.

memMerge Java example

```java
// Create the member merge interaction object.
// This interaction logically merges members together
// into supercession sets.
IxnMemMerge memMerge = new IxnMemMerge(ctx);

// Set the member type as PERSON.
// Member types are listed in mpi_memtype table.
memMerge.setMemType("PERSON");

// Set entity type as Identity (id).
// Entity types are listed in mpi_enttype table.
memMerge.setEntType("id");

// Create two memhead objects, one for surviving
// member another for obsolete member.
// MemHead models the Initiate database table
// mpi_memhead.
MemHead survivor;
MemHead obsolete;

// Set the surviving member's identifiers.
// First Name - Michael Last Name - Howard.
survivor = new MemHead();
survivor.setMemRecno(0);
survivor.setEntRecno(0);
// the following two attributes are required for MemMerge
survivor.setSrcCode("RMC");
survivor.setMemIdnum("112917");

// Set the obsolete member's identifiers.
// First Name - Mike Last Name - Howard.
obsolete = new MemHead();
obsolete.setMemRecno(0);
obsolete.setEntRecno(0);
// the following two attributes are required for MemMerge
obsolete.setSrcCode("RMC");
obsolete.setMemIdnum("300067");

// Execute the member merge interaction.
memMerge.execute(survivor, obsolete);
```

mergeMember web services example

```java
// Create request object using which we pass input
// parameters to the operation.
MemberMergeRequest request = new MemberMergeRequest();
```
// Set the srccode and member id for the surviving member.
request.setSrcCodeSrv(sSrcCode);
request.setMemIdnumSrv(sMemIdnumSrv);

// Set the srccode and member id for the obsolete member.
request.setSrcCodeObs(sSrcCode);
request.setMemIdnumObs(sMemIdnumObs);

// Set the username & password for the operation.
request.setUserName("rwuser");
request.setUserPassword("rwuser");

// Perform the mergeMember operation.
port.mergeMember(request);

Member unmerge

After you merged the two duplicate records of John Bach, you could always unmerge them if needed. Use the IxnMemUnmerge interaction object to unmerge one member from another. The input RowList contains one MemHead segment. The MemHead segment is the obsolete member to be unmerged.

**memUnmerge Java example**

```java
// Create the member unmerge interaction object.
IxnMemUnmerge memUnmerge = new IxnMemUnmerge(ctx);

// Create a member rowlist to hold input member row(s)
MemRowList inpRL = new MemRowList();

// Set the member type as PERSON.
// Member types are listed in mpi_memtype table.
memUnmerge.setMemType("PERSON");

// Set entity type as Identity (id).
// Entity types are listed in mpi_enttype table.
memUnmerge.setEntType("id");

// MemHead models the Initiate database table
// mpi_memhead.
MemHead obsolete = new MemHead();
obsolete.setMemRecno(0);
obsolete.setEntRecno(0);
obsolete.setSrcCode("RMC");
obsolete.setMemIdnum("300067");
inpRL.addRow(obsolete);

// Execute the member unmerge interaction.
memUnmerge.execute(inpRL);
```

**unmergeMember web services example**

```java
// Create a request object using which we pass input
// parameters to the operation.
MemberUnmergeRequest request = new MemberUnmergeRequest();

// Set the srccode and member id of the
// obsolete member, which is to be unmerged.
request.setSrcCodeObs("RMC");
request.setMemIdnumObs("300067");

// Set the username & password for the operation.
request.setUserName("rwuser");
```
request.setUserPassword("rwuser");

// Perform the unmergeMember operation.
boolean cc = port.unmergeMember(request);

**Member delete**

One week before the surgery, John Bach informed Get Well Hospital that his company was thinking of permanently transferring him to a new city and all subsequent care would be performed in this new location. The IT department then decided to temporarily (logically) delete the record of John Bach from the Hub database as he would no longer be returning for service. However, it did not permanently (physically) delete the record, just in case John Bach decided to come back.

The member delete interaction object IxnMemDelete logically deletes a single member from Hub database. This interaction updates the MemStat (member status) column in mpi_memhead table to “D,” and all derived data and rules are deleted for the member. However, the member attribute data is not physically deleted. When physical deletion is required, use the IxnMemDrop interaction.

**memDelete Java example**

// Create a member delete interaction object.
IxnMemDelete memDelete = new IxnMemDelete(ctx);

// Create a member rowlist to hold input member row(s).
MemRowList inpMemRows = new MemRowList();

// Create a MemHead object.
// MemHead models the Initiate database table mpi_memhead.
MemHead memHead = new MemHead();

// Set the identifiers of the member to be deleted.
memHead.setSrcCode("OUTP");
memHead.setMemIdnum("567677");

// Add MemHead into MemRowList.
inpMemRows.addRow(memHead);

// Execute member delete interaction.
memDelete.execute(inpMemRows, KeyType.MEMIDNUM);

**deleteMember web services example**

// Create a request object using which we pass input
// parameters to the operation.
MemberDeleteRequest request = new MemberDeleteRequest();

// Deletes a member created by the exMemPut example.
// Set the source code and member id.
request.setSrcCode("OUTP");
request.setMemIdnum("567677");

// Set the username & password for the operation.
request.setUserName("rwuser");
request.setUserPassword("rwuser");

// Perform the deleteMember operation.
boolean cc = port.deleteMember(request);
Member undelete

As expected by Get Well Hospital, John Bach’s company decided against transferring him to the new city. So the IT department at Get Well Hospital recovered the deleted medical record of John Bach using the member undelete interaction.

Member undelete interaction object IxnMemUndelete returns a logically deleted member to active status by updating the mpi_memhead table’s MemStat (member status) value to “A” (ACTIVE). Derived data is recreated so that the member can take further updates and be the result of a search or a match. Attributes will have the same RecStat values as when the member was deleted.

**memUndelete Java example**

```java
// Create a member undelete interaction object.
IxnMemUndelete memUndelete = new IxnMemUndelete(ctx);

// Create a member rowlist to hold input member row(s).
MemRowList inpMemRows = new MemRowList();

// MemHead models the Initiate database table mpi_memhead.
MemHead memHead = new MemHead();

// Set the identifiers of the member to be undeleted.
memHead.setSrcCode("OUTP");
memHead.setMemIdnum("567677");
inpMemRows.addRow(memHead);

// Execute the member undelete interaction.
memUndelete.execute(inpMemRows, KeyType.MEMIDNUM);
```

**undeleteMember web services example**

```java
// Create a request object using which we pass input
// parameters to the operation.
MemberUndeleteRequest request = new MemberUndeleteRequest();

// Set the member id and source code for the
// undelete member operation.
request.setSrcCode("OUTP"); //SrcRecno = 211 - RMC
request.setMemIdnum("567677");

// Set the username & password for the operation.
request.setUserName("rwuser");
request.setUserPassword("rwuser");

// Perform the undeleteMember operation.
boolean cc = port.undeleteMember(request);
```

Member undo

You can use the IxnMemUndo interaction object to perform a MemUnput (”Member unput” on page 65) or MemUnmerge (”Member unmerge” on page 73) or MemUndelete (”Member undelete”) by passing in an audRecNo. The appropriate interaction is executed, depending on whether the audRecNo is traced back to a record insert or update, a merge, or a delete operation. For details and caveats, refer to the discussion of the other interactions.

**Note:** Potential Overlays cannot be backed out using MemUnput or MemUndo.
This interaction is available for the Java SDK only. For examples, consult the SDK distribution disc.

**Member drop**

A patient named Roberta Salmon was moving out of the city permanently and wanted to transfer her medical records to a new hospital. After handing over her medical records, Get Well Hospital wanted to permanently delete Roberta Salmon’s records. The IT department at Get Well Hospital used the member drop interaction to permanently delete the record.

You can use the IxnMemDrop interaction object to physically delete a member record from the Hub database.

**memDrop Java example**

```java
// Create a member drop interaction object.
IxnMemDrop memDrop = new IxnMemDrop(ctx);

// Create a member rowlist to hold input member row(s).
MemRowList inpMemRows = new MemRowList();

// MemHead models the Initiate database table mpi_memhead.
MemHead memHead = new MemHead();

// Set the identifiers of the member to be dropped.
memHead.setSrcCode("OUTP");
memHead.setMemIdnum("567677");

// Add MemHead into MemRowList.
inpMemRows.addRow(memHead);

// Execute the member drop interaction.
memDrop.execute(inpMemRows, KeyType.MEMIDNUM);
```

**dropMember web services example**

```java
// Create and setup a MemberDropRequest object and send it to the service.
MemberDropRequest request = new MemberDropRequest();

// Set the member id and source code for the dropMember operation.
request.setMemIdnum("567677");
request.setSrcCode("OUTP");

// Set the username & password for the operation.
request.setUserName("rwuser");
request.setUserPassword("rwuser");

// Perform the dropMember operation.
port.dropMember(request)
```

**Member score**

In Good Health Hospital, no patient data can be centrally located. The hospital still wants to match a record from the registration system with records from other systems, such as Get Well Hospital. Good Health Hospital retrieves potential candidates from each system via another method and uses an IxnMemScore interaction object to compare the likelihood of a match of records from these other systems with the registration system record.
You can use IxnMemScore to obtain a matching score between two members (neither of which are required to be known in the Hub database). The output member row list will contain the memhead objects in the same order they were passed in, with the match score set. The match score on the first memhead object will be the first member scored against itself.

**Note:** The engine requires memRecno order for optimized processing. If you are coding a process in which you need to score each member against every other member using the IxnMemScore interaction, create a memRowList representing each candidate. After each execution of IxnMemScore, shift the memRecno for each row in each rowList. Then build the input rowList for the next MemScore execution in order, based on the newly assigned memRecnos.

**memScore Java example**

```java
// Create a member rowlist to hold input member rows
MemRowList inpRowList = new MemRowList();

// MemHead models the Initiate database table mpi_memhead.
MemHead memHead = new MemHead(0);

// Set the identifiers of the member to be scored
MemDate memDate = new MemDate(memHead);
memDate.setAttrCode("BIRTHDT");
memDate.setDateVal("1989-09-30");

MemAttr memSex = new MemAttr(memHead);
memSex.setAttrCode("SEX");
memSex.setAttrVal("M");

MemName memName = new MemName(memHead);
memName.setAttrCode("LGLNAME");
memName.setOnmFirst("Michael");
memName.setOnmLast("de Test");

// Add member info into MemRowList.
inpRowList.addRow(memHead);
inpRowList.addRow(memDate);
inpRowList.addRow(memSex);
inpRowList.addRow(memName);

// MemHead for the second member to score.
MemHead memHead2 = new MemHead(0);

// Set the identifiers of the member to be scored
MemDate memDate2 = new MemDate(memHead2);
memDate2.setAttrCode("BIRTHDT");
memDate2.setDateVal("1998-09-30");
// transpose the 8 and 9

MemAttr memSex2 = new MemAttr(memHead2);
memSex2.setAttrCode("SEX");
memSex2.setAttrVal("M");

MemName memName2 = new MemName(memHead2);
memName2.setAttrCode("LGLNAME");
memName2.setOnmFirst("Michael");
memName2.setOnmLast("de Test");

// Add member info into MemRowList.
inpRowList.addRow(memHead2);
inpRowList.addRow(memDate2);
inpRowList.addRow(memSex2);
inpRowList.addRow(memName2);

// MemHead for the third member to score.
```
MemHead memHead3 = new MemHead(0);

// Set the identifiers of the member to be scored
MemDate memDate3 = new MemDate(memHead3);
memDate3.setAttributeCode("BIRTHDT");
memDate3.setDateVal("1989-09-30");

MemAttr memSex3 = new MemAttr(memHead3);
memSex3.setAttributeCode("SEX");
memSex3.setAttributeVal("M");

MemName memName3 = new MemName(memHead3);
memName3.setAttributeCode("LGLNAME");
memName3.setOnmFirst("Michael");
memName3.setOnmLast("de Test");

// Add member info into MemRowList.
inpRowList.addRow(memHead3);
inpRowList.addRow(memDate3);
inpRowList.addRow(memSex3);
inpRowList.addRow(memName3);

logRowList(inpRowList);

// Create output Row List
MemRowList outRowList = new MemRowList();

// Create MemScore interaction object
IxnMemScore ixnMemScore = new IxnMemScore(ctx);
ixnMemScore.setEntType("id");

// Execute the scoring
if (!ixnMemScore.execute(inpRowList, outRowList))
{
    err("Unable to score Members - " + ixnMemScore.getErrCode() + ": " +
    ixnMemScore.getErrText());
}

// Extract the scores
assertEquals(outRowList.size(), 3);
short lastScore = -1;
RowIterator rowIter = outRowList.rows();

while (rowIter.hasMoreRows())
{
    MemRow memRow = (MemRow) rowIter.nextRow();
    assertTrue(memRow instanceof MemHead);
    MemHead memHead4 = (MemHead) memRow;
    assertTrue(memHead4.getMatchScore() > lastScore);
    lastScore = memHead4.getMatchScore();
}

**memberScore web services example**

// Create and setup a MemberScoreRequest object
// and send it to the service.
MemberScoreRequest request = new MemberScoreRequest();

// Set the identifiers of the member to be score.
request.setMemIdnum("567677");

request.setMemDate memdate = new memDate (memHead);
request.setAttributeCode("BIRTHDT");
request.setDateVal("1989-09-30");

request.setMemAttr memsex = new MemAttr(memHead);
request.setAttributeCode("SEX");
Member get history

IxnMemGetHist is one of the APIs, along with IxnAudSearchHist and IxnRelGetHist, that make up the Historical API feature of the IBM Initiate Master Data Service. The Historical API feature enables you to view data elements as they existed at a particular point in time. Tables in the data model (with the _h suffix) store the historical information. These elements work together to calculate historical data.

The IxnMemGetHist interaction retrieves MemRow objects for the criteria that you specify, enabling you to see, for a given point in time:

- What members belonged to an entity?
- What attribute values did a member have?
- To which entities did a member belong?

Class MemGetHistCriteria provides the input criteria to IxnMemGetHist. Criteria can include entity type, entity record number, member record number, source code, member ID number, get type, and segment attribute and record status filters.
IxnMemGetHist includes audit logging and timer logging with additional data, such as information about objects involved in the interaction.

Refer also to “Audit search history” on page 121 and “Relationship get history” on page 142.

**Note:** In Initiate IBM Initiate Workbench, users must enable the retrieval of historical data for applicable objects. Refer to the IBM Initiate Workbench User’s Guide for configuration information.

For current examples, refer to the SDK distribution disc.

---

**Dictionary interactions**

Dictionary interactions enable you to define the structure and format of the Master Data Engine and Hub database. You can use the dictionary interactions to retrieve and submit dictionary segments to the Master Data Engine.

**Dictionary get**

You can use the IxnDicGet interaction object to retrieve dictionary segments from the Master Data Engine.

**IxnDicGet Java example**

```java
// Create the dictionary get interaction object.
IxnDicGet dicGet = new IxnDicGet(ctx);

// Create dictionary rowlists to hold output row(s).
DicRowList outputRows = new DicRowList();

// SegCode filters are required for this interaction.
// Set the SegCode filter with the segment codes of tables
// we want to dump.
dicGet.setSegCodeFilter("SEGATTR,SRCHEAD,USRHEAD,GRPHEAD,TSKTYPE");

// Set the record status indicators desired.
// The values include (A)ctive, (I)nactive, (D)eleted and (S)hadow.
dicGet.setRecStatFilter(Row.m_RECSTAT_A);

// Execute the dictionary get interaction.
// The output of the interaction is stored in outputRows.
boolean cc = dicGet.execute(outputRows);
```

The resulting rows that are returned are stored in the DicRowList object outputRows.

**getDictionary web services example**

```java
// Create a request object using which we pass input
// parameters to the operation.
DictionaryGetRequest request = new DictionaryGetRequest();

// Set a segment code filter to limit
// output to specific segments.
request.setSegCodeFilter(sSegCodeFilter);

// Set the username & password for the operation.
request.setUserName("rwuser");
request.setUserPassword("rwuser");

// Perform the getDictionary operation.
port.getDictionary(request);
```
**Dictionary put**

The IxnDicPut interaction object can be used to insert or delete dictionary data. You can create or update segments like SrcHead, SegAttr, GrpHead, TskType, and UsrHead, using the dictionary put interaction. The Master Data Engine determines the action to be taken on the dictionary row objects passed in based on the value of their rowInd attribute. Use the SetRowInd() function to set the attribute to one of the following values:

**Note:** The default RowInd is RowInd.INSERT. SetRowInd() needs to be called only if another RowInd behavior is desired.

- **RowInd.IGNORE** – No action is taken
- **RowInd.INSERT** – Dictionary data is inserted into the Hub database
- **RowInd.UPDATE** – Dictionary data is updated in place
- **RowInd.DELETE** – Dictionary data is physically deleted from the Hub database

**Note:** The form of the password used for the creation of an UsrHead object depends on the intended use of the UsrHead. If the UsrHead object is to be used for authentication to perform an interaction, the normal constructor or setUserPass method can be used.

You cannot use hashed passwords to run an interaction.

**IxnDicPut Java example**

```java
// Create the dictionary put interaction object.
IxnDicPut dicPut = new IxnDicPut(ctx);

//create a DicRowList with init size of 2, and grow by 10
DicRowList dicList = new DicRowList(2, 10);

// Create various Dic subclasses. By calling the constructor with a 0, a
// reserved memRecno will be created.

// Create various Dic subclasses
SrcHead srcHead = new SrcHead();
srcHead.setSrcRecno(255);
srcHead.setMemTypeno(1);
srcHead.setSrcCode("TST");
srcHead.setPhyCode("TST");
srcHead.setSrcType("D");
srcHead.setRiCheck("Y");
srcHead.setSrcName("Test Mental Health Center");

SegAttr segAttr = new SegAttr();
segAttr.setAttrRecno(255);
segAttr.setMemTypeno(1);
segAttr.setSegCode("MEMATTR");
segAttr.setAttrCode("TSTATTR");
segAttr.setAttrName("Test Attribute");
segAttr.setAttrLabel("Test Label");
segAttr.setMsFilter("A");

GrpHead grpHead = new GrpHead();
grpHead.setGrpRecno(255);
grpHead.setGrpName("TSTGROUP");
grpHead.setGrpDesc("Tst Group");

TskType tskType = new TskType(255);
tskType.setTskRecno(255);
tskType.setTskType("TST");
tskType.setTskTypeCat("255");
tskType.setTskKind("M");
```

Chapter 3. Basic interactions   81
```java
tskType.setTskTypeLabel("Test Label");
tskType.setIsContent("N");
tskType.setIsLinking("N");

UsrHead usrHead = new UsrHead();
usrHead.setUsrLogin("TstUsr");

// Passwords in the Database are stored
// in encrypted format and need to be
// encrypted before using this DicPut interaction.
usrHead.setCvwPerm("A");
usrHead.setUsrRecno(255);
usrHead.setIxnPerm("A");
usrHead.setSegPerm("W");
usrHead.setAppPerm("A");

// Set the type of operation we are going to do on each
// of the rows we created based on the user's choice.
// Note that the dicPut interaction can be used to insert
// as well as delete dictionary values. The RowInd value
// determines whether it is an insert or delete.
if (bInsFlag)
{
    srcHead.setRowInd(RowInd.INSERT);
    segAttr.setRowInd(RowInd.INSERT);
    tskType.setRowInd(RowInd.INSERT);
    usrHead.setRowInd(RowInd.INSERT);
    grpHead.setRowInd(RowInd.INSERT);
}
else
{
    srcHead.setRowInd(RowInd.DELETE);
    segAttr.setRowInd(RowInd.DELETE);
    tskType.setRowInd(RowInd.DELETE);
    usrHead.setRowInd(RowInd.DELETE);
    grpHead.setRowInd(RowInd.DELETE);
}
dicList.addRow(tskType);
dicList.addRow(segAttr);
dicList.addRow(srcHead);
dicList.addRow(usrHead);
dicList.addRow(grpHead);
dumpMem(dicList);

// Execute the dictionary put interaction.
Boolean cc = dicPut.execute(usr, dicList);

**putDictionary web services example**

// Create Dictionary object which holds
// dictionary information.
Dictionary dictionary = new Dictionary();

// SrcHeadWs array which holds the header information.
SrcHeadWs[] srcHead = new SrcHeadWs[2];

// Set various fields for the first SrcHeadWs segment.
srcHead[0] = new SrcHeadWs();
srcHead[0].setSrcRecno(iRecno.intValue());
srcHead[0].setMaxLen(10);
srcHead[0].setMemTypeno(1);
srcHead[0].setMinLen(1);
srcHead[0].setPhyCode("TST"+iRecno);
srcHead[0].setPoCmpCode("");
srcHead[0].setPoScore((short)0);
srcHead[0].setRiCheck("Y");
srcHead[0].setSrcCode("TST"+iRecno);"
srcHead[0].setSrcName("Test Mental Health *+iRecno");
srcHead[0].setSrcType("D");
srcHead[0].setRecStat("A");

// Set the type of operation indicating
// whether to Add or Delete row.
srcHead[0].setRowInd(sRowInd);

// Set various fields for the second SrcHeadWs segment.
// with SrcRecno + 1 to be passed.
Integer iRecno2 = new Integer(iRecno.intValue() + 1);
srcHead[1] = new SrcHeadWs();
srcHead[1].setSrcRecno(iRecno2.intValue());
srcHead[1].setMemTypeno(1);
srcHead[1].setSrcCode("TST"+iRecno2);
srcHead[1].setPhyCode("TST"+iRecno2);
srcHead[1].setRecStat("A");
srcHead[1].setSrcType("D");
srcHead[1].setRiCheck("Y");
srcHead[1].setSrcName("Test Mental Health *+iRecno2");

// Set the type of operation, this value
// indicates whether to Add or Delete row.
srcHead[1].setRowInd(sRowInd);

// Add the source header information to the dictionary.
dictionary.setSrcHead(srcHead);

// Create UsrHeadWs object which holds
// user header information.
UsrHeadWs[] usrHead = new UsrHeadWs[1];
usrHead[0] = new UsrHeadWs();
usrHead[0].setUsrLogin("TSTUsr"+iRecno);

// Passwords in the database are stored
// in encrypted format and need to be
// encrypted before using putDictionary operation.
usrHead[0].setUsrPassEncrypt("TSTUsr"+iRecno);
usrHead[0].setCvwPerm("A");
usrHead[0].setUsrRecno(iRecno.intValue());
usrHead[0].setIxnPerm("A");
usrHead[0].setSegPerm("W");

// Set the type of operation.
usrHead[0].setRowInd(sRowInd);

// Add the user header information to the dictionary.
dictionary.setUsrHead(usrHead);

// Create a request object using which we pass input
// parameters to the operation.
DictionaryPutRequest request = new DictionaryPutRequest();

// Set username & password for the operation.
request.setUserName("rwuser");
request.setUserPassword("rwuser");

// Set the Dictionary object in the request.
request.setDictionary(dictionary);

// Perform the putDictionary operation.
port.putDictionary(request);
Set user password

You can use the IxnUsrSetPass interaction object to change a password for an IBM Initiate Master Data Service software user.

**IxnUsrSetPass Java example**

```java
// Create user Head object with passed-in username and password.
UsrHead usr = new UsrHead(userName, oldPassword);

// Create a context with host info passed in.
Context ctx = new Context(hostName, hostPort, usr);

// Create user head object.
UsrHead usrHead = new UsrHead();
usrHead.setUsrLogin(userName);

// Create the set user password interaction object.
IxnUsrSetPass setPass = new IxnUsrSetPass(ctx);

// Execute the interaction.
boolean cc = setPass.execute(usrHead);
```

**putUserPassword web services example**

```java
// Create a request object using input
// parameters passed to the operation.
UserPasswordPutRequest request = new UserPasswordPutRequest();

// Set the current Password and UserName.
request.setUserName(userName);
request.setUserPassword(oldPass);

// Set the new Password.
request.setNewUserPassword(newPass);

// Perform the putUserPassword operation.
port.putUserPassword(request);
```

Get user information

The IxnUsrGetInfo interaction retrieves dictionary data about users from the Hub. This Get interaction has an option to take a single usrhead input row list with a UsrLogin value and retrieve all related segments for this user.

It outputs zero or more of the following segments based on the segcode filter:
USRHEAD, GRPHEAD, GRPXIXN, GRPXSEG, GRPXCVW, GRPXAPP

The user's last successful login timestamp is updated by specifying the keyword LOGIN with the setArgs method. Refer to “setArgs” on page 143.

```java
IxnUsrGetInfo ixn = new IxnUsrGetInfo(context);
ixn.setSegCodeFilter("USRHEAD");
ixn.setArgs("LOGIN");
```

By default, the timestamp appears in MM/DD/YYYY HH:MM format.

For current examples, refer to the SDK distribution disc.
Audit interactions

Audit get

The IxnAudGet interaction retrieves audit data from IBM Initiate Master Data Service Hub.

For current examples, refer to the SDK distribution disc (Java only; the WS SDK does not include an example for this interaction.)

Audit search history

To enable users to view data elements as they existed at a particular point in time, the Historical API feature was developed for the IBM Initiate Master Data Service. It comprises three API calls:

- IxnAudSearchHist
- IxnMemGetHist
- IxnRelGetHist

and new tables in the data model (with the _h suffix) to store the historical information. These elements work together to calculate historical data.

The IxnAudSearchHist interaction enables you to use specific criteria to search within AudHead objects for a specific audit record number for a member, entity, or relationship.

- Class AudSearchHistCriteria provides the input criteria used by AudSearchHist. Criteria can include entity type, entity record number, member record number, source code, member ID number, relationship type number, relationship linkage number, any segment attribute filters, and the history type.
- The objects can correspond to member attribute, relationship attribute, relationship changes and entity composition. The AudHead object is used with the IxnMemGetHist and IxnRelGetHist interactions to find how the member, entity, or relationship looked at a given point in time, or at every change through a range of dates.

IxnAudSearchHist includes audit logging and timer logging with additional data, such as information about objects involved in the interaction.

Refer also to “Member get history” on page 112 and “Relationship get history” on page 142.

Note: In Initiate IBM Initiate Workbench, users must enable the retrieval of historical data for applicable objects. Refer to the IBM Initiate Workbench User’s Guide for configuration information.

For current examples, refer to the MemGetHist and RelGetHist examples on the SDK distribution disc.
Task interactions

Task search

You can view various tasks generated by the Master Data Engine using the Task Search interaction. The task search interaction object IxnTskSearch is used to return a list of task rows that fit a list of search criteria.

Task types include:
- Type no: 1 - Potential Overlay
- Type no: 2 - Potential Duplicate
- Type no: 3 - Potential Linkage
- Type no: 4 – Review Identifier
- Type no: 5 – PreMerge
- Type no: 6 – HasShadow

For instance, you could search for Potential Overlay tasks (type “1”) with a score less than 99.

The task search interaction is slightly different from the other interactions in that it uses MemXtsk and EntXTsk objects for search input.

IxnTskSearch Java example

```java
// Create the task search interaction object.
IxnTskSearch tskSearch = new IxnTskSearch(ctx);

// Create a member rowlist to hold input member row(s).
MemRowList inputRowList = new MemRowList();

// Create a member rowlist to hold output row(s).
MemRowList outputRowList = new MemRowList();

// Create a MemHead object. MemHead models
// Initiate database
// table mpi_memhead.
// A specific member's task can be retrieved
// by specifying the MemRecno.
MemHead memHead = new MemHead();
memHead.setMemRecno(0);
memHead.setEntRecno(0);
memHead.setSrcRecno(0);
memHead.setMemIdnum("");
inputRowList.addRow(memHead);

// Search for a Task Type - Potential Overlay.
// Task types are listed in mpi_tsktype.
MemXtsk memXtsk = new MemXtsk();
memXtsk.setTskTypeno(1);
memXtsk.setMaxScore((short) 99);
inputRowList.addRow(memXtsk);

// Set the segCodeFilter to limit the output to
// specific segments if required.
tskSearch.setSegCodeFilter("MEMHEAD,MEMXTSK,ENTXTSK");

// Set the record status indicators desired.
// The values include (A)ctive, (I)nactive, (D)eleted and (S)hadow.
tskSearch.setRecStatFilter("A");
```
// Display the contents of the RowList, row by row. 
dumpRows(inputRowList);

// Set entity type as Identity (id) 
// Entity types are listed in mpi_enttype table. 
tskSearch.setEntType("id");

// Execute the task search interaction. 
boolean status = tskSearch.execute(inputRowList, outputRowList);

The outputRowlist will contain the segments for the members that matched the search criteria.

searchTask web services example

Note: Example code for this operation is also available in C# (.NET). Refer to the following files for the complete code: “exTskSearch.cs.”

// Create a request object using which we pass input // parameters to the operation. 
TaskSearchRequest request = new TaskSearchRequest();

// Create a member object to hold member information. 
Member member = new Member();

// Create a MemHeadWs object to hold member name. 
MemHeadWs memHead = new MemHeadWs();
member.setMemHead(memHead);

// Create a MemXtskWs object to hold member task. 
MemXtskWs[] memXtsk = new MemXtskWs[1];
memXtsk[0] = new MemXtskWs();

// Find tasks that have a score less than the max of -99. 
memXtsk[0].setMaxScore((short) -99);

// Set the MemXtskWs object in the member. 
member.setMemXtsk(memXtsk);

// Create the EntXtskWs object. 
// Set the minimum score. 
EntXtskWs[] entXtsk = new EntXtskWs[1];
entXtsk[0] = new EntXtskWs();
entXtsk[0].setMinScore((short) 99);

// Set the EntXtskWs object to the member. 
member.setEntXtsk(entXtsk);

// Set the Member object to the request. 
request.setMember(member);

// Set a segment code filter to limit // output to specific segments. 
request.setSegCodeFilter("MEMHEAD,MEMNAME,MEMXTSK,ENTXTSK");

// Set the record status indicators desired. // The values include (A)ctive, (I)nactive, // (D)eleted and (S)hadow. 
request.setRecStatFilter("A");

// Set the username & password for the operation. 
request.setUserName("rwuser");
request.setUserPassword("rwuser");

// Set entity type as Identity (id). // Entity types are listed in mpi_enttype table. 
request.setEntType("id"); //hh or id.

// Set the member type as PERSON. // Member types are listed in mpi_memtype table. 
request.setMemType("PERSON");

// Perform the searchTask operation. 
port.searchTask(request);
Task get

You can use the task get interaction object, IxnTskGet, to retrieve all members that have issues with the member you are “getting.” For instance, John Bach has issues with a member named Jonathan Bach because he had attributes very similar to those of John Bach within the same source system. This issue manifests itself as a Potential Duplicate task. The IxnTskGet interaction object would retrieve this task and any other members sharing this task. A task get is usually performed following a task search to retrieve the complete task set.

IxnTskGet Java example

// Create the task get interaction object.
IxnTskGet tskGet = new IxnTskGet(ctx);

// Create member rowlists to hold input
// and output row(s).
MemRowList tskRowList = new MemRowList();
MemRowList memOutRowList = new MemRowList();

// Create usr Head object with username and password.
UsrHead usrHead = new UsrHead("rwuser", "rwuser");

// Create a MemHead object. MemHead models
// Initiate database.
MemHead memHead = new MemHead();
memHead.setMemRecno(memRecno);
memHead.setEntRecno(0);
memHead.setSrcRecno(0);
memHead.setMemIdnum("");
tskRowList.addRow(memHead);

// Set a segCodeFilter which will limit the output to just
// specific segments.
tsGet.setSegCodeFilter("MEMHEAD,MEMXTSK,ENTXTSK");
tskGet.setSegAttrFilter("");

// Set the record status indicators desired.
// The values include (A)ctive, (I)nactive, (D)eleted and (S)hadow.
tsGet.setRecStatFilter("A");

// Set entity type as Identity (id).
// Entity types are listed in mpi_enttype table.
tsGet.setEntType("id");

// Execute the task get interaction.
boolean cc = tskGet.execute(usrHead, tskRowList, memOutRowList);

The member segments will be stored in memOutRowList. Remember that in order to get any member segments in the results you must add them to the segcode filter.

getTask web services example

// Create a request object using which we pass input
// parameters to the operation.
TaskGetRequest request = new TaskGetRequest();

// Set the member record number.
// Member record numbers are listed in mpi_memxtsk table.
request.setMemRecno(new Integer(167));

# Set a segment code filter to limit
// output to specific segments.
request.setSegCodeFilter("MEMHEAD,MEMNAME,MEMXTSK,ENTXTSK");
// Set the username & password for the operation.
request.setUserName("rwuser");
request.setUserPassword("rwuser");

// Set entity type as Identity (id).
// Entity types are listed in mpi_enttype table.
request.setEntType("id");

// Set the member type as PERSON.
// Member types are listed in mpi_memtype table.
request.setMemType("PERSON");

// Perform the getTask operation.
port.getTask(request);

**Task get 2**

This is similar to the traditional Task get interaction, but IxnTskGet2 populates the new MEMRECNOS array in all task types. The traditional Task get interaction leaves this field null.

- For mpi_memxtsk and mpi_entxtsk records, MEMRECNOS is a list of all memrecnos for the task type.
- For mpi_idtxtsk records, this is a list of the MEMRECNOS associated to an individual task record.

Task get 2 "rolls up" mpx_idtxtsk records with the same TSKRECNO into a set so that only one implementation-defined (custom) task object (idtxtsk) is returned for a taskset. This differs from the traditional IxnTskGet, which will present all mpi_idtxtsk records.

**Task put**

One day the orthopedic department staff unknowingly updated an old record with new attributes, essentially replacing the old record of John Bach with a new person named Roberta Salmon. Attributes such as name, phone number and e-mail address were completely different while the source ID and member ID were the same. In such a situation you could insert a Potential Overlay task using a task put interaction.

You can insert tasks into the Hub database using the task put interaction object IxnTskPut.

The Master Data Engine determines the action to be taken on the task row objects passed in, based on the value of their RowInd attribute. You can use the setRowInd() method to set the attribute to one of the following values:

- **RowInd.IGNORE** – No action is taken.
- **RowInd.INSERT** – Task data is inserted into the Hub database.
- **RowInd.UPDATE** – Task data is updated in place.
- **RowInd.DELETE** – Task data is physically deleted from the Hub database.

**IxnTskPut Java example**

```java
// Create the task put interaction object.
IxnTskPut tskPut = new IxnTskPut(ctx);

// Create member rowlists to hold input rows.
MemRowList tskRL = new MemRowList();
```
// Create a DicStore object.
DicStore dic = null;
try {
    dic = new DicStore(ctx);
} catch (Exception e) {
    System.err.println("ERROR:Dictionary creation failed with an Exception");
    System.exit(1);
}

int tr = 1; //Potential Overlay
TskType taskType = dic.getTskTypeByRecno(tr);

// Create a MemHead object. MemHead models
// Initiate database
// table mpi_memhead.
MemHead memHead = new MemHead();
memHead.setMemIdnum("500567");
memHead.setSrcCode("OUTP");

// Setup the input row.
MemXtsk memXtsk = new MemXtsk(memHead);

// SetRecno which is used to group the memXtsk of all members
// that belong to the same Task.
int iSetRecno = 1;
memXtsk.setRecno(iSetRecno);
memXtsk.setTskType(taskType);
memXtsk.setMemRecno(90);
memXtsk.setSrcCode("OUTP");
memXtsk.setMaxScore((short) 80);
memXtsk.setTskStatno(3);
memXtsk.setUsrRecno(1);
memXtsk.setRecCtime(new GregorianCalendar().getTime());
memXtsk.setRecMtime(new GregorianCalendar().getTime());
memXtsk.setRecCtimeBefore(new GregorianCalendar().getTime());
memXtsk.setRecCtimeSince(new GregorianCalendar().getTime());
memXtsk.setRowInd(RowInd.INSERT);

// Add the row.
tskRL.addRow(memXtsk);

// Setup the input row.
MemHead memHead2 = new MemHead();

// Add the tskHead row to the input RowList.
memHead2.setMemIdnum("100439");
memHead2.setSrcCode("OUTP");
MemXtsk memXtsk2 = new MemXtsk(memHead2);

// Use the same SetRecno value as above to group task.
memXtsk2.setRecno(iSetRecno);
memXtsk2.setTskType(taskType);
memXtsk2.setMemRecno(89);
memXtsk2.setSrcCode("OUTP");
memXtsk2.setMaxScore((short) 80);
memXtsk2.setTskStatno(3); //Deferred
memXtsk2.setUsrRecno(1);
memXtsk2.setRecCtime(new GregorianCalendar().getTime());
memXtsk2.setRecMtime(new GregorianCalendar().getTime());
memXtsk2.setRecCtimeBefore(new GregorianCalendar().getTime());
memXtsk2.setRecCtimeSince(new GregorianCalendar().getTime());
memXtsk2.setRowInd(RowInd.INSERT);
memXtsk2.setRowInd(RowInd.INSERT);

// Add the row.
tskRL.addRow(memXtsk2);

// Display the task row values.
System.out.println(memXtsk);
System.out.println(memXtsk2);

// Execute the task put interaction.
boolean status = tskPut.execute(tskRL, PutType.INSERT_UPDATE);

The PutType parameter can be one of the following:
- PutType.INSERT_UPDATE - If the task data does not exist, create it; or if the
  task data already exists, update it
- PutType.INSERT_ONLY - If the task data does not exist, create it; or if the task
  data already exists, abort the interaction
- PutType.UPDATE_ONLY - If the task data already exists, update it; or if the task
  data does not exist, abort the interaction

putTask web services example

// Set the RecNo.
// setRecno is used to group the memXtskWses of
// all members that belong to the same Task.
int iSetRecno = 1;

// Use memHeadWses to create the Task.
MemHeadWs memHeadPre = getMemHeadWs(port, sSrcCode, sMemIdnum);
System.out.println("*** Do Task Put ****");

// Create a member object to hold member information.
Member member = new Member();

// Create the memXtskWs Array.
MemXtskWs[] memXtsk = new MemXtskWs[1];

// Create the memXtskWses object and set the attributes.
memXtsk[0] = new MemXtskWs();
memXtsk[0].setSetRecno(iSetRecno);
memXtsk[0].setMemRecno(memHeadPre.getMemRecno());
memXtsk[0].setSrcCode(memHeadPre.getSrcCode());
memXtsk[0].setMaxScore((short) -80);
memXtsk[0].setTskTypeno(1); //Potential Overlay.
memXtsk[0].setTskStatno(3); //Deferred.
memXtsk[0].setUsrRecno(1);
member.setMemXtsk(memXtsk);

// Create a request object using which we pass
// input parameters to the operation.
TaskPutRequest request = new TaskPutRequest();

// Set the type of put.
request.setPutType("INSERT_ONLY");

// Set the username & password for the operation.
request.setUserName("rwuser");
request.setUserPassword("rwuser");

// Set the Member object to the request.
request.setMember(member);

// Perform the putTask operation.
port.putTask(request);
The PutType parameter can be one of the following:

- **PutType.INSTERT_UPDATE** - If the task data does not exist, create it; or if the task data already exists, update it
- **PutType.INSTERT_ONLY** - If the task data does not exist, create it; or if the task data already exists, abort the operation
- **PutType.UPDATE_ONLY** - If the task data already exists, update it; or if the task data does not exist, abort the operation

**Task put 2**

Similar to the traditional Task put interaction, but IxnTskPut2 allows the use of the MEMRECNOs array to specify members associated to an individual task. The traditional Task put interaction leaves this field null. This interaction is used for custom tasks; use the idtxtsk object to get and set the text and hint fields. Refer to the IBM Initiate Master Data Service Feature Quick Start Guide for more information.

- For mpi_memxtsk and mpi_entxtsk records, a list of tasks with an associated SETRECNO are created in the database.
- For mpi_idtxtsk records, a single task is created with the member associations persisted in the mpi_tskxmem table.

Custom task data should, at a minimum, include a task type (tasktypeno) and the member record number (memrecno) for the member record with which a given custom task will be associated.

One thing to note about using IxnTskPut2 to update tasks is that you first need to retrieve the task from the hub using the tskrecno. In the past, you could update a task using the memrecno because a member could only have one task. With the introduction of custom tasks, a member can now have multiple tasks, therefore, the tskrecno is the key, rather than the memrecno.

**Task put by criteria**

IxnTskPutByCriteria assigns the specified owner to every task in the system that matches the specified criteria.

For current examples, refer to the SDK distribution disc.

**Task search count**

The IxnTskSearchCount interaction returns the count of every task in the system that matches the specified criteria, and optionally the count of each tag type.

For current examples, refer to the SDK distribution disc.

**Enterprise Identity Assignment (EIA) interactions**

In Inspector, this data appears in the Linkage views. The EIA get/search/put interactions are very similar to the task versions of these interactions. The main difference is they deal with EIA segments instead of task segments. The EIA data shows the history of member entity assignments. EIA records are created when an Enterprise ID is assigned to a member (either due to Master Data Engine processing or a task resolution) and when the status of a member is updated.
EIA search

You can use the EIA search interaction object IxnEIASearch to search for EIA records that meet a combination of member and/or EIA criteria.

**IxnEiaSearch Java example**

```java
// Create the EIA search interaction object.
IxnEiaSearch eiaSearch = new IxnEiaSearch(ctx);

// Create a member rowlist to hold input member row(s)
MemRowList inputRowList = new MemRowList();

// Create a member rowlist to hold output row(s)
MemRowList outputRowList = new MemRowList();

// Create a MemHead object. This object models the
// Initiate database table mpi_memhead
MemHead memHead = new MemHead();
inputRowList.addRow(memHead);

// Search for a eia type 7 - Merge
MemXeia memXeia = new MemXeia();
memXeia.setEiaTypeno(7);

// Find records that are less than max score of 9.9
memXeia.setMaxScore((short) 99);
inputRowList.addRow(memXeia);

// Set a segment code filter to limit
// output to specific segments.
eiaSearch.setSegCodeFilter("MEMHEAD, MEMXEIA");

// Set the record status indicators desired.
// The values include (A)citive, (I)nactive, (D)eleted and (S)hadow.
eiaSearch.setRecStatFilter("A");

// Display the contents of the RowList, row by row.
dumpRowsList(inputRowList);

// Set entity type as Identity (id)
// Entity types are listed in mpi_enttype table.
eiaSearch.setEntType("id");

// Execute the EIA search interaction.
boolean status = eiaSearch.execute(inputRowList, outputRowList);
```

The outputRowList will contain the segments for the members that matched the search criteria.

**searchEia web services example**

```java
// Create a request object using which we pass
// input parameters to the operation.
EiaSearchRequest request = new EiaSearchRequest();

// Create a member object to hold member information.
Member member = new Member();

// Create the MemHead object and set it to the member.
MemHeadWs memHead = new MemHeadWs();
member.setMemHead(memHead);

// Create a MemXeia object and set the EiaTypeno.
MemXeiaWs[] memXeia = new MemXeiaWs[1];
memXeia[0] = new MemXeiaWs();
```
/ Search for Merge.
memXeia[0].setEiaTypeno(10);
member.setMemXeia(memXeia);

// Create EntXeia with the EiaTypeno set to search for
EntXeiaWs[] entXeia = new EntXeiaWs[1];
entXeia[0] = new EntXeiaWs();

// Search for AutoLink-SS
entXeia[0].setEiaTypeno(1);
entXeia[0].setMaxScore((short)250);
member.setEntXeia(entXeia);

// Set the Member object to the request.
request.setMember(member);

// Will return all MemXeia & EntXeia rows
// that meet the set of criteria
request.setSegCodeFilter("MEMHEAD,MEMNAME,MEMXEIA,ENTXEIA");

// Set the record status indicators desired.
// The values include (A)ctive, (I)nactive,
// (D)eleted and (S)hadow.
// Only Active segments will return
request.setRecStatFilter("A");

// Set the username & password for the operation.
request.setUserName("rwuser");
request.setUserPassword("rwuser");

// Set entity type as Identity (id).
// Entity types are listed in mpi_enttype table.
request.setEntType("id"); //hh or id.

// Set the member type as PERSON.
// Member types are listed in mpi_memtype table.
request.setMemType("PERSON");

// Perform the getApplicationInfo operation.
port.searchEia(request);

EIA get

You can use the EIA Get interaction object IxnEiaGet to retrieve EIA records.

IxnEiaGet Java example

// Create the EIA get interaction object.
IxnEiaGet eiaGet = new IxnEiaGet(ctx);

// Create member rowlists to hold input.
MemRowList eiaRowList = new MemRowList();

// Create a member rowlist to hold output row(s)
MemRowList eiaOutRowList = new MemRowList();

// Create a MemHead object. MemHead models
// Initiate Initiate database table mpi_memhead.
MemHead memHead = new MemHead();
memHead.setSrcCode("RMC");
memHead.setMemIdnum("300067");
eiaRowList.addRow(memHead);

// Set a segment code filter to limit
// output to specific segments.
eiaGet.setSegCodeFilter("MEMHEAD,MEMXEIA,ENTXEIA");

// Set entity type as Identity (id)
// Entity types are listed in mpi_enttype table.
eiaGet.setEntType("id");

// Set the record status indicators desired.
// The values include (A)cctive, (I)nactive, (D)eleted and (S)hadow.
eiaGet.setRecStatFilter("A");

// Execute the interaction. If any EIA records are found
// related to the input member, they will be returned in the
// output RowList.
// GetType.ASMEMBER- specifies that only individual
// members will be retrieved.
// KeyType.MEMIDNUM - specifies the srcCode/memIdnum is the
// retrieval key.
boolean status = eiaGet.execute(eiaRowList, eiaOutRowList,
                   GetType.ASMEMBER, KeyType.MEMIDNUM);

The member segments will be stored in eiaOutRowList. Remember that in order to
get any member segments in the results, you must add them to the segcode filter.

getEia web services example
// Create a request object using which we pass input
// parameters to the operation.
EiaGetRequest request = new EiaGetRequest();

// Set the source code and member Id
// for the operation.
request.setSrcCode(sSrcCode);
request.setMemIdnum(sMemIdnum);

// Set a segment code filter to limit
// output to specific segments.
request.setSegCodeFilter(sSegCodeFilter);

// Set the username & password for the operation.
request.setUserName("rwuser");
request.setUserPassword("rwuser");

// Set entity type as Identity (id).
// Entity types are listed in mpi_enttype table.
request.setEntType("id"); //hh or id.

// Creation/modification dates/times will not be filled in.
request.setAudMode("AUDNONE");

// Perform the getEia operation.
port.getEia(request);

Remember that you must add member segments to the segcode filter in order to
get them in the results.

EIA put
You can use the EIA put interaction object IxnEiaPut to populate EIA data.

IxnEiaPut Java example
// Create the interaction object.
IxnEiaPut eiaPut = new IxnEiaPut(ctx);

// Get the MemExeia records into the MemRowList.
MemRowList eiaRowList = doEiaGet(ctx);
// Create member rowlist to hold input rows.
MemRowList eiaInputRowList = new MemRowList();

// Create a ClassTest object to use as a filter
// to allow only MemXeia rows when obtaining a RowIterator
ClassTest memXeiaTest = null;
try {
    memXeiaTest = new ClassTest("madison.mpi.MemXeia");
} catch (Exception e) {
    System.out.println("Error " + e.getMessage());
}

// Get a RowIterator using a classTest of memXeia.
// Traverse through the RowList to
// get the MemXeia rows and update each record.
RowIterator memXeiaIter = eiaRowList.rows(memXeiaTest);
while (memXeiaIter.hasMoreRows()) {
    MemXeia memXeia = (MemXeia) memXeiaIter.nextRow();
    if (memXeia.getEiaType().equalsIgnoreCase("Delete")) {
        memXeia.setEiaStatno(2); //mark as examined-OK
    }
    memXeia.setTskTypeno(2); //Potential Duplicate
    memXeia.setTskStatno(3); //Deferred
    memXeia.setRowInd(RowInd.UPDATE);
    eiaInputRowList.addRow(memXeia);
}

// Execute the interaction.
boolean cc = eiaPut.execute(usr, eiaInputRowList, PutType.UPDATE_ONLY);

putEia web services example

// Create a member object to hold member information.
Member member = new Member();

// Set the MemXeiaWs object in the member.
member.setMemXeia(memXeia);

// Create a request object using which we pass input
// parameters to the operation.
EiaPutRequest request = new EiaPutRequest();

// Set put type to update.
request.setPutType("UPDATE_ONLY");

// Set the username & password for the operation.
request.setUserName("rwuser");
request.setUserPassword("rwuser");

// Set the Member object to request.
request.setMember(member);

// Perform the putEia operation.
port.putEia(request);
Relationship interactions

Relationship get

You can use the relationship get interaction object IxnRelGet to retrieve relationships for a list of entities from the Hub database. The relationship get interaction retrieves relationship links associated with a specified list of RelLinknos.

For current examples, refer to the SDK distribution disc.

Relationship put

You can use the IxnRelPut interaction object to create relationships between entities and to submit modified relationship rows to the Master Data Engine. Criteria include the following:

- Each relationship submitted must have a corresponding relTypeno that matches an existing relType.
- Both entRecnoLeft and entRecnoRight must be set on each relLink. The relLinkList must contain only instances of relLink.
  - relLinks with relationship attribute beans (relAttrBean) are supported. relAttrBeans are defined dynamically with the appropriate fields based on the segHead and relSegAttr definitions.
- Each entRecno specified must be of the entity type defined by the corresponding entTypenoLeft and entTypenoRight fields. In addition, each relationship created must honor the relType.multiplicity (the one-to-one relationship, one-to-many relationship, or many-to-many relationship) setting.
- If the relLinkno field is not explicitly set, the Master Data Engine will assign a unique relLinkno and attempt to persist the relLink as a new instance. If relLinkno is set, the Master Data Engine treats this as an update. For a new instance, the relLinknos can be retrieved from the relLinks returned by getResult().

The newly persisted relLinks and associated relAttrBeans will be created, deleted, and updated based on the tracking information maintained by the relLink.

If a relationship already exists in the database, an error code of EEXISTS is returned.

For current examples, refer to the SDK distribution disc.

Relationship search

Use the IxnRelSearch interaction to retrieve relationships for a list of entities from the Hub database. This interaction retrieves relationship links associated with a specified list of entRecnos that contains only instances of long-representing entRecnos. The specified relTypeno must match an existing relType.

The dirType determines how the input entRecno list is processed. dirTypes are defined as follows:

- DIR_TYPE_LEFT: entRecnos are matched against relLink.entRecno.Left
- DIR_TYPE_RIGHT: entRecnos are matched against relLink.entRecno.Right
DIR_TYPE_BOTH: entRecnos are matched against both relLink.entRecno.Left and relLink.entRecno.Right

Each entRecno in the entRecnoList must be the appropriate entity type based on the specified dirType and corresponding entType for the given relType. Consider the following example.

```
IF dirType is DIR_TYPE_LEFT and relType.entTypenoLeft is N
THEN each entRecno in the list must be of an entity with an entTypeno of N
```

Successful execution of the interaction results in relLinks associated with the specified list of entRecnos. Each relLink represents a single relationship between entities.

For current examples, refer to the SDK distribution disc.

**Relationship search count**

The IxnRelSearchCount interaction retrieves the relationship count for a list of entities from the Hub database. This interaction differs from IxnRelGet in that only the number of relationship links is returned by IxnRelSearchGet rather than the actual relationships.

For current examples, refer to the SDK distribution disc.

**Relationship path to root**

Use the IxnRelPathToRoot interaction to retrieve a path from one node (entRecno at the starting point) to the root of the tree in a hierarchical relationship. The result is a list of all entRecnos on the path.

A hierarchical relationship is defined as follows:
- Multiplicity is one-to-one or one-to-many.
- The relationship type is directional.
- The entity type is the same on both sides.

For current examples, refer to the SDK distribution disc.

**Relationship delete**

The member delete interaction object IxnRelDelete deletes the specified relationship row in the Master Data Engine database, thus removing relationships between the entities. This interaction keys off the list of relLinknos input.

For current examples, refer to the SDK distribution disc.

**Relationship move**

This new interaction added atomically deletes a relationship between a target entity and another entity, and then creates a new relationship between the target
entity and a new entity. For example target entity A has an old relationship with B
which needs to move to entity C. The IxnRelMove interaction causes a new
relationship between A and C to be created and the old relationship between A
and B to be deleted.

**Relationship search task**

Use the IxnRelTskSearch interaction to retrieve relationship tasks from the Hub
database.

For current examples, refer to the SDK distribution disc.

**Relationship get task**

The IxnRelTskGet interaction retrieves relationship tasks that have specified
tskRecnos.

relXtskCriteria is used as the input criteria to the IxnRelTskSearch interaction. If
keyType.entRecno is specified, then all tasks for that entity and its members are
searched. If keyType.entRecno or keyType.memIdnum is specified, then only tasks
for the member are searched.

For current examples, refer to the SDK distribution disc.

**Relationship resolve task**

Use the IxnRelTskResolve interaction to resolve relationship tasks.

For current examples, refer to the SDK distribution disc.

**Relationship delete task**

The IxnRelTskDelete interaction deletes the specified relationship tasks.

For current examples, refer to the SDK distribution disc.

**Relationship get history**

IxnRelGetHist is one of the APIs, along with IxnAudSearchHist and
IxnMemGetHist, that make up the Historical API feature of the IBM Initiate Master
Data Service. The Historical API feature enables you to view data elements as they
existed at a particular point in time. Tables in the data model (with the _h suffix)
store the historical information. These elements work together to calculate
historical data.

The IxnRelGetHist interaction retrieves RelLink objects for the criteria that you
specify. The RelLinks represent a relationship at a point in time, based on the given
AudHead, and enable you to see, for a given point in time:

* What relationships belonged to an entity?
* What attribute values did a relationship have?
* To which entities did a relationship belong?
Class RelGetHistCriteria provides the input criteria to IxnRelGetHist. Criteria can include relationship linkage number, entity type, entity record number, and relationship type number.

IxnRelGetHist includes audit logging and timer logging with additional data, such as information about objects involved in the interaction.

Refer also to “Audit search history” on page 121 and “Member get history” on page 112.

Note: In Initiate IBM Initiate Workbench, users must enable the retrieval of historical data for applicable objects. Refer to the IBM Initiate Workbench User’s Guide for configuration information.

For current examples, refer to the SDK distribution disc.

---

Tag interactions

**Tag get**

The IxnTagGet interaction retrieves a tag from the Master Data Engine.

For current examples, refer to the SDK distribution disc.

**Tag put**

Use the IxnTagPut interaction to create or update a tag instance in the Master Data Engine.

For current examples, refer to the SDK distribution disc.

**Tag reset**

IxnTagReset deletes all tag instances for the tag type and reprocesses all tasks to determine whether the tag applies to each task.

For current examples, refer to the SDK distribution disc.

**Tag delete**

The IxnTagDelete interaction removes a tag instance from the Master Data Engine.

For current examples, refer to the SDK distribution disc.

---

Groups interactions

Information about LDAP groups are stored in the Hub database and need to be synchronized with the LDAP server when groups are added, deleted or updated programmatically.

**groups update**

IxnGroupsUpdate creates or updates group DN information. For current examples, refer to the SDK distribution disc.
groups sync
Use IxnGroupsSync to execute group synchronization with the LDAP server. This performs the same operation that the Refresh Groups button does in the IBM Initiate Workbench Groups view. For current examples, refer to the SDK distribution disc.

groups get info
IxnGrpGetInfo retrieves user dictionary data from the Hub. For current examples, refer to the SDK distribution disc.

### Handler interactions

When engine callout is enabled, you can pass information to your custom handlers using the following interaction methods.

**setArgs**

Pass information to the handler specific to the current interaction. Any arguments passed from the client application through the Ixn.setArgs() method will be available to the handler through the IService.getArgs() method. These interaction-specific arguments can be used to pass additional information to the handler at run-time.

Usage: `Ixn.setArgs(String);`

**getUsrErrCode**

This retrieves any handler-specific error code that may have occurred when the handler was executed.

Usage: `usrErrCode = Ixn.getUsrErrCode();`

**getErrText**

This retrieves any handler-specific error message string that may have occurred when the handler was executed.

Usage: `String s = Ixn.getErrText();`

### IBM Initiate Flexible Search Interactions

Two new interactions support IBM Initiate Flexible Search: IxnMemTextSearch and IxnMemTermSearch. Both are in the madison.mpi.search package.

### Searching with the API

The Java and REST APIs are used to perform searches of the IBM Initiate Flexible Search index. It is interaction-based like the rest of the APIs for the IBM Initiate Master Data Service.

### About this task

Two new interactions support IBM Initiate Flexible Search: IxnMemTextSearch and IxnMemTermSearch. Both are in the madison.mpi.search package.
Procedure

1. To compile and run a program using IBM Initiate Flexible Search, put the MAD_ROOTDIR/lib/sdk/madapi.jar file into your CLASSPATH.
2. See the IBM Initiate Java SDK Javadoc Information for full details of the API.
3. Build queries as needed. For more information, refer to the "Query language" topic within the IBM Initiate Flexible Search User’s Guide.

Java API index search classes and examples

The IxnMemTextSearch and IxnMemTermSearch interactions support IBM Initiate Flexible Search.

See the IBM Initiate Java SDK Javadoc Information for complete details.

IxnMemTextSearch

Provides full-text search capability against a configured IBM Initiate Master Data Service full-text index. The API provides interfaces to either retrieve matches by using a traditional row list approach, or by using a new composite class structure. The row list API is provided as a convenience for those who want to utilize full-text search without significantly changing existing application code. The composite class structure is an instance of MemTextSearchResult, which handles the details of mapping data records to their corresponding member record identifiers. This structure also allows the retrieval of stored values that are associated with documents within the index.

IxnMemTermSearch

Provides term search capability against a configured IBM Initiate Master Data Service full-text index. Search criteria is provided in the form of field name and word prefix. The field name must be a configured entity in the index and the word prefix must be at least two characters in length. Results of the search are obtained by using the getResult method, which returns a List of MemTermSearchResults.

Example: simple search

This example performs a simple search with a query specified on the command line.

```java
package your.search.example;

import madison.mpi.Context;
import madison.mpi.GetType;
import madison.mpi.MemName;
import madison.mpi.MemRow;
import madison.mpi.MemRowList;
import madison.mpi.search.IxnMemTextSearch;

public class SimpleExample
{
    public static void main( String[] args)
    {
        //Query string comes from the command line.
        String query = args[0];

        //Substitute values appropriate for your MDS.
        Context ctx = new Context("localhost", 16000, "system", "system");

        //Search for members that match.
        //We'll only look at the name attributes and active values.
        IxnMemTextSearch ixn = new IxnMemTextSearch( ctx);
        ixn.setSegCodeFilter("MEMHEAD,MEMNAME");
    }
}
```
Example: simple search using MemTextSearchResult

This example does a simple search with a query specified on the command line but uses the MemTextSearchResult object for results. Use the MemTextSearchResult interaction to organize results by member, include the full-precision score in results, or when you want access to the stored fields.

The full-precision score might differ from the score returned by MemHead.getMatchScore() because the full-precision score is a Java double and the MemHead score is a Java short. Stored fields are useful when you do not want to retrieve all member details in the search results. You can configure the index to hold a brief summary of a member in a stored field. The summary field can then be used to display the search results.

```java
package your.search.example;
import java.util.Map;
import madison.mpi.Context;
import madison.mpi.GetType;
import madison.mpi.MemHead;
import madison.mpi.search.IxnMemTextSearch;
import madison.mpi.search.MemTextHitResult;
import madison.mpi.search.MemTextSearchResult;

public class SimpleExample2
{
    public static void main( String[] args)
    {
        //Query string comes from the command line.
        String query = args[0];

        //Substitute values appropriate for your MDS.
        Context ctx = new Context("localhost", 16000, "system", "system");

        //Search for members that match.
        //We will not retrieve any attributes.
        IxnMemTextSearch ixn = new IxnMemTextSearch( ctx);
        ixn.setSegCodeFilter("MEMHEAD");
        ixn.setRecStatFilter("A");

        //Execute the interaction.
        if( !ixn.execute( query, GetType.ASMEMBER ) )
        { throw new RuntimeException("Ixn failed. " + ixn.getErrText());
        }

        //Display the results.
        for( int i = 0; i < rows.size(); ++i ){
            MemRow row = rows.rowAt(i);
            if( !(row instanceof MemName) ) continue;

            MemName name = (MemName) row;
            System.out.println( name.getSrcCode() + "/" + name.getMemIdnum()
                               + "\" + name.getAttrCode()
                               + "\" + name.getOnmFirst() + "\" + name.getOnmLast());
        }
    }
}
```
throw new RuntimeException("Ixn failed. " + ixn.getErrText());
}

//Loop for each member in the result
MemTextSearchResult result = ixn.getResult();
for( MemTextHitResult hit: result.getHits() ){

    //Display member identifier and score
    MemHead head = hit.getMemHead();
    System.out.println( head.getSrcCode() + "/" + head.getMemIdnum() + " " +
                       hit.getScore());

    //Display any stored values.
    Map storedValues = hit.getStoredValues();
    for( Map.Entry entry: storedValues.entrySet() ){
        System.out.println( entry.getKey() + "=" + entry.getValue());
    }

}

Example: performing a term search

This example uses the IxnMemTermSearch interaction to search for matching terms. Term searches must specify a field to search. All matching terms that start with the specified prefix are returned, up to a specified maximum. Note that the prefix must be at least two characters in length, otherwise no results will be returned.

The term search is useful for implementing "instant" functionality where matching terms can be shown while you are typing a query.

package your.search.example;

import madison.mpi.Context;
import madison.mpi.search.IxnMemTermSearch;
import madison.mpi.search.MemTermSearchResult;

public class SimpleExample3
{
    public static void main( String[] args)
    {
        //Field name, prefix and max come from the command line.
        String fieldName = args[0];
        String prefix = args[1];
        int max = Integer.parseInt( args[2]);

        //Substitute values appropriate for your MDS.
        Context ctx = new Context("localhost", 16000, "system", "system");

        //Limit the number of answers we get.
        IxnMemTermSearch ixn = new IxnMemTermSearch( ctx);
        ixn.setMaxRows( max);

        //Execute the interaction.
        if( !ixn.execute( fieldName, prefix) )
            throw new RuntimeException("Ixn failed. " + ixn.getErrText());

        //Display the results.
        for( MemTermSearchResult match: ixn.getResult() )
            System.out.println( match.getTerm()+" " + match.getFrequency());
    }
}
Chapter 4. Java Callout handlers

You might want to send notifications or API calls to external systems when certain events have occurred within the Master Data Engine, such as entity linking and unlinking, member data modifications, and member searches.

This feature enables you to create custom callout handlers (also called “handlers”) to manage your special business needs, such as:

- Callouts to a third-party API
- Applying conditional security for certain types of interactions
- Sending event notifications to an external message queue, email address, or file system

All these tasks are accomplished by registering and deploying custom Java handlers, leveraging the classes found in madapi.jar and madhandlers.jar.

Each handler is registered and deployed on a specific Master Data Engine instance. At the time of registration, you specify the interaction types that will trigger it, as well as the callback type. The callback type determines when, in relation to the interaction, the handler is invoked.

Custom handlers are created using one of the Initiate SDKs. Once handlers are created and packaged, they are registered and deployed using IBM Initiate Workbench. Consult the IBM Initiate Workbench User’s Guide for instructions on packaging, registering and deploying callout handlers.

Creating custom handlers

About this task

Your custom handler will be executed in anticipation of or in response to an interaction occurring within the Master Data Engine or an entity manager. What the handler does, specifically, is up to you. Examples include:

- Sending a brief message to a pager or email address
- Posting a message in a messaging queue
- Calling a third-party rules engine API
- Applying special security to allow or disallow the interaction
- Updating data in the Hub database

There are two steps required to create a custom event handler:

Procedure

1. Implement the IHandler interface or extend a class that does, such as madison.handler.HandlerExtBase.

   Note: Use the madison.handler.HandlerExtBase class as the base class for the handler whenever possible. This class is available in the madhandlers.jar, located in the [engine]/lib/engine/contrib directory. The HandlerExtBase class adds the ability to log messages to the standard Master Data Engine log file.

   public class FileHandler : HandlerExtBase
2. Use the base methods to initialize and execute the handler.
   - The init() method has access to the Handler args and is responsible for storing those args as instance member variables. Initialize any global instance resources, such as database connections, in init().
   - The iHandler.invoke() method has access to both the CallbackType and the IService instance. Use the IService instance to get access to the ixnType and the input and output rows.

   **Note:** If your handler extends "madison.handler.HandlerExtBase," the preIxn(), postIxn() and/or entMng() methods can be implemented instead of invoke().

   - The preIxn() method is executed immediately preceding the action specified by the interaction code (such as ALL, MEMGET, MEMPUT, ENTMNG). Interaction codes that trigger this method are determined when the handler is registered, rather than in code.
   - The postIxn() method is executed immediately following the action specified by the interaction code. Interaction codes that trigger this method are determined when the handler is registered, rather than in code.
   - The entMng() method is executed immediately after the entity manager has run.
   - Use the shutdown() method to clean up any global instance resources.

**Results**

Consult the Javadoc HTML documentation for more detailed information on using the handler classes and methods.

**Example handler to write event information to a file**

The examples below include logic for both preIxn() and postIxn() methods instead of using the invoke() method. When we register this handler, we would specify the ALL CallbackType so that when our desired interaction type occurs (for example, a MEMPUT), both of our handler methods would be executed.

**Java handler example**

```java
package com.initiatesystems.hub.handler;

import java.io.BufferedWriter;
import java.io.File;
import java.io.FileWriter;
import java.io.PrintWriter;
import java.util.Map;
import madison.handler.CallbackHandlerException;
import madison.handler.HandlerExtBase;
import madison.handler.IService;
import madison.mpi.Context;
import madison.mpi.RowIterator;
import madison.mpi.RowList;

public class FileHandler extends HandlerExtBase {
    // Class-specific members used as keys in the handlerArgs to identify the
    // pre and post-ixn filename(s)
    private static final String ARG_PREFILENAME = "preFileName";
    private static final String ARG_POSTFILENAME = "postFileName";

    // Instance-specific members initialized during the overridden init() method
```
private String preFileName = null;
private String postFileName = null;

public void init(Context ctx, String handlerArgs) {
    Map argsMap = parseArgs(handlerArgs);
    String madHomeDir = System.getenv("MAD_HOMEDIR");

    // Create the pre- and post- files in the MAD_HOMEDIR directory
    if (argsMap.containsKey(ARG_PREFILENAME)) {
        preFileName = madHomeDir + File.separator +
            (String)argsMap.get(ARG_PREFILENAME);
    }
    if (argsMap.containsKey(ARG_POSTFILENAME)) {
        postFileName = madHomeDir + File.separator +
            (String)argsMap.get(ARG_POSTFILENAME);
    }
}

protected void writeFile(String fileName, RowList rowList) {
    PrintWriter out = null;
    try {
        out = new PrintWriter(new BufferedWriter(new
            FileWriter(fileName)));
        for (RowIterator i = rowList.rows(); i.hasMoreRows(); ) {
            out.println(i.nextRow().toString());
        }
    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        if (out != null) {
            try {
                out.close();
            } catch (Exception e) {

            }
        }
    }
}

public void preIxn(IService service) throws CallbackHandlerException {
    if (preFileName != null) {
        writeFile(preFileName, service.getInpMemRowList());
    }
}

public void postIxn(IService service) throws CallbackHandlerException {
    if (postFileName != null) {
        writeFile(postFileName, service.getOutMemRowList());
    }
}
Example handler to test Best Match Linking

This code block represents a Java callout handler that tests for EID changes during Best Match Linking processing. It demonstrates how to use the EIA type to determine EID changes when reprocessing non-trusted source members.

In order to allow callout handlers to determine whether the EID change is significant, additional data beyond that being changed is passed to the callout handler. The new data consists of all of the EntXeia rows for the audRecno being processed, including previously existing rows with a row indicator value of ignore. Using this data, combined with EntXeia rows created during the reprocessing of the current non-trusted source record allows the callout handler to determine whether the EID change was an insignificant change, a move to another entity, or whether the record remained as a singleton after being flagged for reprocessing. This logic is contained in the method mapMemberMoves in the code.

See comments within the code for additional information.

Java handler example

```java
package com.initiatesystems.hub.link;

import java.util.Map;
import java.util.HashMap;
import madison.handler.CallbackHandlerException;
import madison.handler.HandlerExtBase;
import madison.handler.IService;
import madison.mpi.MemHead;
import madison.mpi.MemRowList;
import madison.mpi.RowIterator;
import madison.mpi.EntIque;
import madison.mpi.Row;
import madison.mpi.EntXeia;
import madison.mpi.UsrHead;
import madison.mpi.IxnMemPut;
import madison.mpi.RowInd;
import madison.mpi.PutType;
import madison.mpi.MemMode;
import madison.mpi.MatchMode;
import madison.mpi.Context;
import madison.util.ClassTest;

/**
 * A Java callout handler for BML testing.
 */
public class HandlerBML extends HandlerExtBase {

private static final String TEST_NAME = "TestLTEIDChangeCallout";
private static final String ENT_TYPE = "LT_BML";
private static final int EIATYPE_AUTOLINKSS = 1;
private static final int EIATYPE_AUTOLINKMS = 2;
private static final int EIATYPE_REPROCESS = 14;

/**
 * An enumeration to track move types.
 */
private static enum MoveType {
    NoChange,
    MoveToAnotherEntity,
    Singleton,
    ChangeToNewEntity
}

public class HandlerBML extends HandlerExtBase {
    private static final String TEST_NAME = "TestLTEIDChangeCallout";
    private static final String ENT_TYPE = "LT_BML";
    private static final int EIATYPE_AUTOLINKSS = 1;
    private static final int EIATYPE_AUTOLINKMS = 2;
    private static final int EIATYPE_REPROCESS = 14;

    /**
     * An enumeration to track move types.
     */
    private static enum MoveType {
        NoChange,
        MoveToAnotherEntity,
        Singleton,
        ChangeToNewEntity
    }

    public class HandlerBML extends HandlerExtBase {
        private static final String TEST_NAME = "TestLTEIDChangeCallout";
        private static final String ENT_TYPE = "LT_BML";
        private static final int EIATYPE_AUTOLINKSS = 1;
        private static final int EIATYPE_AUTOLINKMS = 2;
        private static final int EIATYPE_REPROCESS = 14;

        /**
         * An enumeration to track move types.
         */
        private static enum MoveType {
            NoChange,
            MoveToAnotherEntity,
            Singleton,
            ChangeToNewEntity
        }
    }
```
* on reprocess.
/*
EntityMove,
/**
 * Moving from the trusted source to a singleton
 * only.
 */
NewEntity,
/**
 * New members moving into an entity on reprocess.
 */
NewMember
}

@Override
protected void entMng(IService service) throws CallbackHandlerException {
    info("entMng [" + getName() + "]");
    if (!service.getEntType().equals(ENT_TYPE)) {
        // not our desired entity type so bail quick
        return;
    }

    // we're looking for reprocessed members from
    // the TestLTEIDChangeCallout test
    MemRowList inputRows = service.getInpMemRowList();
    MemHead memhead = null; // we need the memhead
    that got updated
    EntIque entique = null; // we need the entique
    row to know it's reprocess
    RowIterator rowIterator = inputRows.rows();
    // loop over the rows until we find the two we're
    looking for or there aren't any more
    while (rowIterator.hasMoreRows() && (memhead ==
        null || entique == null)) {
        Row row = rowIterator.nextRow();
        if (row instanceof MemHead) {
            memhead = (MemHead) row;
        } else if (row instanceof EntIque) {
            entique = (EntIque) row;
        }
    }
    // if we didn't find the two rows we're looking for
    // we're done
    if (memhead == null || entique == null) {
        warn("Could not find memhead or entique row.");
        return;
    }
    // we're only looking for Reprocess callouts
    if (!entique.getWrkCode().equals("R")) {
        return;
    }
    String memIdnum = memhead.getMemIdnum();
    // we're only looking for member updates from
    // our test
    if (!memIdnum.startsWith(TEST_NAME)) {
        return;
    }
    // examine the EIA rows to determine any moves
    Map<Long, MoveType> moves = mapMemberMoves(inputRows);
    // get the memRecno from the entique so we can
    // check the member we're testing
    long memRecno = entique.getMemRecno();
    MoveType moveType = moves.get(memRecno);

Chapter 4. Java Callout handlers 109
// memIdnum indicates which test we're working on
if (memIdnum.equals(TEST_NAME + "A1")) {
    if (moveType == MoveType.NoChange) {
        putMember(service.getContext(), "F1");
    } else {
        error("Expect unit test failure due to invalid moveType.*");
    }
} else if (memIdnum.equals(TEST_NAME + "A2")) {
    if (moveType == MoveType.EntityMove) {
        putMember(service.getContext(), "F1");
    } else {
        error("Expect unit test failure due to invalid moveType.*");
    }
} else if (memIdnum.equals(TEST_NAME + "A3")) {
    if (moveType == MoveType.NewEntity) {
        putMember(service.getContext(), "F1");
    } else {
        error("Expect unit test failure due to invalid moveType.*");
    }
} else if (memIdnum.equals(TEST_NAME + "A4")) {
    if (moveType == MoveType.NoChange) {
        putMember(service.getContext(), "F1");
    } else {
        error("Expect unit test failure due to invalid moveType.*");
    }
} else if (memIdnum.equals(TEST_NAME + "A5")) {
    if (moveType == MoveType.EntityMove) {
        putMember(service.getContext(), "F1");
    } else {
        error("Expect unit test failure due to invalid moveType.*");
    }
} else if (memIdnum.equals(TEST_NAME + "A6")) {
    if (moveType == MoveType.NewEntity) {
        putMember(service.getContext(), "F1");
    } else {
        error("Expect unit test failure due to invalid moveType.*");
    }
}

/**
 * A method for determining if an EID change happened
 * for a given member during a Best Match Linking entity
 * update. When a trusted source member is updated,
 * all of the related members are forced into singleton
 * entities and reprocessed by the entity manager. This
 * method examines the entity manager callout data to
 * determine what type of entity move occurred when
 * the non-trusted source members were reprocessed.
 * <p/>
 * This method should be passed the input rows provided
 * to the callout when the EntIque row's wrkCode has an
 * "R" (reprocess) value. This code is NOT the same
 * value present in the service arguments in the callout.
 * Using this method in the callout on non-reprocessed
 * members will result in invalid results.
 * <p/>
 * @param inputRows a MemRowList provided by the
 * entity manager callout that
 * contains EntXeia rows for a
*/
protected Map<Long, MoveType> mapMemberMoves(MemRowList inputRows) {
    if (isDebugEnabled()) {
        debug("Mapping member moves");
    }
    // we need to map the entxeia rows
    Map<String, EntXeia> eiaMap = new HashMap<String, EntXeia>();
    RowIterator rowIterator = inputRows.rows(new ClassTest(new EntXeia()));
    long memRecno;
    int eiaTypeno;
    while (rowIterator.hasMoreRows()) {
        EntXeia row = (EntXeia) rowIterator.nextRow();
        // map has a compound key of memRecno-eiatype
        memRecno = row.getMemRecno();
        eiaTypeno = row.getEiaTypeno();
        if (isDebugEnabled()) {
            debug("Mapping entxeia row " + memRecno + " - " + eiaTypeno);
        }
        eiaMap.put(memRecno + " - " + eiaTypeno, row);
    }
    // track movements
    Map<Long, MoveType> moves = new HashMap<Long, MoveType>();
    MoveType moveType;
    // walk the map to determine the move types
    for (String key : eiaMap.keySet()) {
        String[] keyParts = key.split("-");
        memRecno = Long.valueOf(keyParts[0]);
        eiaTypeno = Integer.valueOf(keyParts[1]);
        moveType = moves.get(memRecno);
        if (eiaTypeno == EIATYPE_AUTOLINKSS || eiaTypeno
            == EIATYPE_AUTOLINKMS) {
            // if eiaTypeno in {1,2} AutoLink
            // we should have a corresponding eiaTypeno 14
            if (eiaMap.containsKey(memRecno + " - " + EIATYPE_REPROCESS)) {
                // we can figure out the entity moves
                long startEntRecno = eiaMap.get(memRecno + " - " + EIATYPE_REPROCESS).getPrevEntRecno();
                long endEntRecno = eiaMap.get(key).getSupEntRecno();
                if (startEntRecno == endEntRecno) {
                    moves.put(memRecno, MoveType.NoChange);
                } else {
                    moves.put(memRecno, MoveType.EntityMove);
                }
            } else {
                // a link without a reprocess is simply a new
                // member joining this entity
                moves.put(memRecno, MoveType.NewMember);
            }
        } else if (moveType == null && eiaTypeno == EIATYPE_REPROCESS) {
            // if eiaTypeno == 14 TSUpdReprocess and we don't
            // have a moveType then we can assume we're a
// singleton
moves.put(memRecno, MoveType.NewEntity);
}
}
if (isDebugEnabled()) {
    debug("Move map: " + moves);
}
return moves;

/**
 * Creates a member with the specified tinyIdnum.
 * @param context used to create the member
 * @param tinyIdnum the member's tiny ID number
 * @throws CallbackHandlerException if the member can't be created
 */
protected void putMember(Context context, String tinyIdnum) throws CallbackHandlerException {
    if (isDebugEnabled()) {
        debug("Creating member with tinyIdnum: " + tinyIdnum);
    }

    // no user/pw available from entMng
    UsrHead usrHead = new UsrHead("system", "system");
    IxnMemPut ixnMemPut = new IxnMemPut(context);
    ixnMemPut.setEntType(ENT_TYPE);
    MemRowList inpMemRows = new MemRowList();
    MemRowList outMemRows = new MemRowList();
    MemHead memHeadPut = new MemHead();
    memHeadPut.setSrcCode(ENT_TYPE + tinyIdnum.charAt(0));
    memHeadPut.setMemIdnum(TEST_NAME + tinyIdnum);
    memHeadPut.setRowInd(RowInd.INSERT);
    inpMemRows.addRow(memHeadPut);
    if (!ixnMemPut.execute(usrHead, inpMemRows, outMemRows, PutType.INSERT_ONLY, MemMode.COMPLETE, MatchMode.DONOTHING)) {
        throw new CallbackHandlerException("Unable to put Member - " + ixnMemPut.getErrCode() + ": " + ixnMemPut.getErrText());
    }
}

Accessing beans in callout handlers

You can access the data passed in the IxnRelPut, IxnRelDelete, and so on, operations in both the PRE and POST callout hook points.

The data for most of the newer interactions such as Relationships, History, Tags, and so forth, are not treated as "rows" but as beans.” They are serialized in a slightly different way to support bean concepts.

For example, the RelLink bean has collections of related RelAttrBeans directly accessible from the class. The data is not found in the inpMemRowList/outMemRowlist properties of the IService that is passed to the callout (like in...
IxnMemGet, IxnMemPut, and so on.). Instead, the data is stored in the argsExt (for input) and resultArgsExt (for output) properties of the IService.

The data types in the argsExt and resultArgsExt properties correspond to the arguments on the execute() method of the ixn. For example, the IxnRelPut has one input arg that is a List<RelLink>. The index in the Object[] corresponds to the order of the args on the execute method. In this case the List<RelLink> is arg 0. The argsExt for the input are always an Object[] to account for multiple parameters. To get the arg from the IService, invoke this code in PRE ixn callout handler:

```java
List<RelLink> inputRelLinks = (List<RelLink>) ((Object[]) service.getArgsExt())[0];
```

The type of the IxnRelPut.getResult() is also a List<RelLink>, so invoke this code to get the results:

```java
List<RelLink> outputRelLinks = (List<RelLink>) ((Object[]) service.getResultArgsExt())[0];
```

The RelLinker interaction type is only used during entity management (the ENTMNG callout hook point). This hook point are not invoked during a standard interaction (for example, IxnRelPut). This is analogous to the EntMngMem (EntLinker) interaction type also being invoked during entity linking. The data in the RelLinker callout is a ListRelLink that includes all the RelLinks and RelXtsks that were created, modified or deleted during Relationship linking. Refer to the Javadoc HTML documentation for the RowBeanData class for more detailed information. Invoke this code in the ENTMNG for a RelLinker callout handler to get access to the data:

```java
List<RowBeanData> relStuff = (List) service.getArgsExt();
```

Logging messages from the handler code

You can enable standard logging by using the following methods.

**Java message logging**

With the Java SDK, you enable logging by extending the handler from madison.handler.HandlerExtBase. For more information, refer to "Creating custom handlers" on page 105.

The following methods are provided by HandlerExtBase:

```java
protected void trace(String msg)
protected void trace(String msg, Throwable t)

protected void debug(String msg)
protected void debug(String msg, Throwable t)

protected void info(String msg)
protected void info(String msg, Throwable t)

protected void warn(String msg)
protected void warn(String msg, Throwable t)

protected void error(String msg)
protected void error(String msg, Throwable t)

protected void fatal(String msg)
```

Chapter 4. Java Callout handlers 113
protected void fatal(String msg, Throwable t)
protected boolean isTraceEnabled()
protected boolean isDebugEnabled()
protected boolean isInfoEnabled()
protected boolean isWarnEnabled()
protected boolean isErrorEnabled()
protected boolean isFatalEnabled()

The custom handler code that extends HandlerExtBase can invoke any of these methods to log information at the level specified. All logging is displayed in the standard engine log if the appropriate logging levels are enabled.

Note: While System.out.println() statements executed in a handler can be output in the mpinet.out file, it is recommended that you use logging capabilities.

MemRowListUtils method with RowLists

MemRowListUtils is a helper method for working with RowLists in engine callout code. For more information, refer to the java madapi.jar in the core SDK.

Compiling Java handler class files

About this task

Compile all handler class files by using madapi.jar and, optionally, madhandlers.jar. Put classes and any extra jar files required at runtime in a unique deployment directory.

For example:
C:\handlers\deploy
   \com\initiatesystems\hub\handler\ this matches the class's package
       FileHandler.class
   MyOtherHandler.class
   \lib\myhelper.jar

Do not include madapi.jar or madhandlers.jar in the deployment directory tree.

Error Handling

CallbackHandlerException can be thrown from invoke() (or from preIx(), postIx(), and entMng() if you are extending HandlerExtBase). If it is thrown, any remaining handlers will not execute for a given CallbackType/IxnCode.

Note: CallbackHandlerException is the only checked exception class for handlers; if an unchecked exception is thrown, such as NullPointerException, it will be wrapped with CallbackHandlerException.

In addition:
• Any exception thrown from a preIx handler will prevent the interaction from executing.
• Any exception thrown from an entMng Handler will be logged by the Entity Manager.
• Any exception thrown from a postIx handler will be logged by the Master Data Engine and returned to the client.
The handler can specify a custom error message and/or usrErrCode when creating the exception. The client application can get the message with the Ixn.getErrText() method and the usrErrCode with the Ixn.getUsrErrCode() method.

---

**Deploying event handlers**

**About this task**

Once handlers are created and packaged, they are registered and deployed using IBM Initiate Workbench. Consult the [IBM Initiate Workbench User’s Guide](#) for instructions on performing steps 1-4 below:

**Procedure**

1. Enable engine callouts in the Master Data Engine instance.
2. Package the handler
3. Register the handler
4. Deploy the handler
5. Test the handler

**Results**

Note: Registration and deployment of handlers is normally done through IBM Initiate Workbench; registration and deployment can also be done with madconfig. It is important to note that if you choose to register your handlers via madconfig, the dictionary will get out of sync when the Master Data Engine configuration is deployed from IBM Initiate Workbench or vice versa.

Note: All classes referenced by registered handlers (except madapi.jar and madhandlers.jar for Java handlers) must be available in the deployment package.

**Test the handler**

Initiate Systems recommends that you deploy new handlers on a test Master Data Engine instance before you deploy them into the production instance. After deploying them, use a client application, such as Inspector, to perform the interaction that will trigger the handler, such as a Member Get or Search.

View the instance engine log file (located in the `hub_instance_path\log` directory) to verify that the handler was executed. Setting MAD_DEBUG or MAD_TRACE in the `hub_instance_path\inst\mpinet_hub_instance_name\conf\log4j.xml` file will write more information to the log file. For instructions on setting log levels, consult the “Configuring Environment Variables and Settings” chapter in the [IBM Initiate Master Data Service Engine Installation Guide](#).

**JMX Monitoring of Java handlers**

Statistics are available at the global callback level as well as for each handler at each callbackType/ixnCode registration point. Available statistics include the number of executions and average, maximum and minimum execution time. In addition, the handler registration data is available for each handler instance.

JMX MBeans registered:

```
com.initiatesystems.<hubInstanceName>.Callouts - Global callback statistics
```
com.initiatesystems.<hubInstanceName>.Callouts.<callbackType>-
<ixnCode>.<handlerCode><handlerNo> - Handler instance statistics

Callout related JMX MBeans are registered and unregistered during handler deployment and undeployment.
Appendix A. Overview of API Updates by Version

This section lists changes to the interactions and operations in versions of the SDK offerings. For more information, refer to the release notes for the applicable version.

**Version 10.0**

New classes are provided within the ESOA and REST APIs for creating a standard or custom task as well as for getting, searching, and updating tasks. For more information, see the IBM Initiate Java SDK Javadoc or the online REST documentation.

**Version 9.7**

The SDK for Microsoft .NET has been removed from this guide and placed into its own document, *IBM Initiate Master Data Service SDK Reference for Microsoft .NET*.

The member attribute description (mpi_edtelem.elements value) is now exposed in the Java and Web Services SDKs. Two new MemAttr methods were added to the SDK: getElemDesc and setElemDesc. getElemDesc returns the MemAttr description through enumeration from mpi_edtelem, if one exists. A virtual field, "elemdesc" is added to MemAttr for the element description. The MemAttr.elemdesc field is populated during MemGet.

Two new interactions support IBM Initiate Flexible Search:

- **IxnMemTextSearch**
  Provides full text search capability against a configured IBM Initiate Master Data Service full text index. The API provides interfaces to either retrieve matches by using a traditional row list approach or through a new composite class structure. The row list API is provided as a convenience for developers who want to use full text search without significantly changing existing application code.

- **IxnMemTermSearch**
  Provides term search capability against a configured IBM Initiate Master Data Service full text index. Search criteria is provided in the form of field name and word prefix.

**Version 9.5**

Added Change Request support and cautionary text for the interactions that support it (IxnMemPut, IxnMemPutBulk, IxnRelPut, IxnRelMove, IxnRelDelete).

Added the new IxnRelMove interaction.

**Version 9.2**

Added MemUnput (IxnMemUnput) and MemUndo (IxnMemUndo) interactions to Java and .NET SDKs. Also added a new method, setEntPrior, to the IxnMem class to enable developers to set the entity management priority when calling member put interactions (MemPut, MemPutBulk, and MemPutQual).
Version 9.0

Added Task Get 2 (IxnTskGet2) and Task Put 2 (IxnTskPut2) interactions to Java, .NET and web services SDKs.

For IxnHead and IxnStat objects, totSndSize and totRcvSize are now persisted as kilobytes instead of bytes.

Version 8.7

.NET SDK

- Added new interactions and classes to support Historical Search feature
  - class IxnAudSearchHist searches AudHead objects for specific criteria.
  - class AudSearchHistCriteria provides the input criteria used by
    AudSearchHist. Criteria can include entity type, entity record number,
    member record number, source code, member ID number, relationship type
    number, relationship linkage number, any segment attribute filters, and the
    history type.
  - enum HistType defines the valid types of history search and retrieval.
  - class IxnMemGetHist retrieves the MemRow for the specified criteria that
    represents how a member or entity looked at a given point in time.
  - class MemGetHistCriteria provides the input criteria to MemGetHist. Criteria
    can include entity type, entity record number, member record number, source
    code, member ID number, get type, and segment attribute and record status
    filters.
  - class IxnRelGetHist retrieves relationship linkage objects for the specified
    criteria.
  - class RelGetHistCriteria provides the input criteria to RelGetHist. Criteria can
    include relationship linkage number, entity type, entity record number, and
    relationship type number.
  - Added IxnMemPutQual interaction to combine some capabilities of MemSearch,
    MemMatch, and MemPut interactions by evaluating records to be added against
    records already in the database and scoring thresholds, with the intent of
    preventing redundant members.
  - Added LOGIN keyword and arguments to IxnUsrGetInfo to support Last Login
    Timestamp feature for IBM Initiate Workbench

Java SDK

- Added new interactions and classes to support Historical Search feature
  - class IxnAudSearchHist searches AudHead objects for specific criteria.
  - class AudSearchHistCriteria provides the input criteria used by
    AudSearchHist. Criteria can include entity type, entity record number,
    member record number, source code, member ID number, relationship type
    number, relationship linkage number, any segment attribute filters, and the
    history type.
  - enum HistType defines the valid types of history search and retrieval.
  - class IxnMemGetHist retrieves the MemRow for the specified criteria that
    represents how a member or entity looked at a given point in time.
- class MemGetHistCriteria provides the input criteria to MemGetHist. Criteria can include entity type, entity record number, member record number, source code, member ID number, get type, and segment attribute and record status filters.
- class IxnRelGetHist retrieves relationship linkage objects for the specified criteria.
- class RelGetHistCriteria provides the input criteria to RelGetHist. Criteria can include relationship linkage number, entity type, entity record number, and relationship type number.
  - Added IxnMemPutQual interaction to combine some capabilities of MemSearch, MemMatch, and MemPut interactions by evaluating records to be added against records already in the database and scoring thresholds, with the intent of preventing redundant members.
  - Added LOGIN keyword and arguments to IxnUsrGetInfo to support Last Login Timestamp feature for IBM Initiate Workbench

**Web services SDK**

- Added IxnMemPutQual interaction to combine some capabilities of MemSearch, MemMatch, and MemPut interactions by evaluating records to be added against records already in the database and scoring thresholds, with the intent of preventing redundant members.
- Added LOGIN keyword and arguments to IxnUsrGetInfo to support Last Login Timestamp feature for IBM Initiate Workbench

**Version 8.5**

**.NET SDK**

- Added support for the following interactions:
  - TagDelete removes a tag instance from the Master Data Engine.
  - TagPut creates or updates a tag instance in the Master Data Engine.
  - TagGet retrieves a tag from the Master Data Engine.
  - TagReset resets the given tag types. All tag instances for the tag type are deleted and all tasks are reprocessed to determine if the tag applies to each task.
  - TskPutByCriteria assigns the specified owner to every task in the system that matches the specified criteria (for example, assigns all Potential Duplicate tasks to User 1).
  - TskSearchCount returns the count of every task in the system that matches the specified criteria and, optionally, the count of each tag type.
  - GroupsUpdate creates or updates group DN information.
  - GroupsSync is used to perform group synchronization with the LDAP server.
- Added getTags() methods.
- Added a new TskKey class to support input to TagGet.
- Added new TagGet bean.
- Modified the TskSearch interaction with the option to search using tags.
- Modified the EntXtsk and MemXtsk objects:
  - getUsrRecno method returns 1 (system user) if the OwnerType is not “U”
  - setUsrRecno method sets both the OwnerType (to “U”) and OwnerRecno properties
- Added new properties and corresponding get/set accessors for OwnerType and OwnerRecno
- Added new OwnerName property. OwnerName represents the GrpName if OwnerType is set to “G” and represents UsrLogin if OwnerType is “U”.

Java SDK

- Added support for the following interactions:
  - TagDelete removes a tag instance from the Master Data Engine.
  - TagPut creates or updates a tag instance in the Master Data Engine.
  - TagGet retrieves a tag from the Master Data Engine.
  - TagReset resets the given tag types. All tag instances for the tag type are deleted and all tasks are reprocessed to determine if the tag applies to each task.
  - TskPutByCriteria assigns the specified owner to every task in the system that matches the specified criteria (for example, assigns all Potential Duplicate tasks to User 1).
  - TskSearchCount returns the count of every task in the system that matches the specified criteria and, optionally, the count of each tag type.
  - GroupsUpdate creates or updates group DN information.
  - GroupsSync is used to perform group synchronization with the LDAP server.
- The CmpSpec.java method was added to the cmpspec classes to support getting and setting the comparison mode (cmpmode).
- Added the following constants:
  - MPI_CMPMODE_MATCH_LINK = 1;
  - MPI_CMPMODE_SEARCH = 2;
  - MPI_CMPMODE_BOTH = 3;
- Added the methods getCmpMode (returns integer) and setCmpMode (accepts integer).
- Added getTags() methods.
- Added a new TskKey class to support input to TagGet.
- Added new TagGet bean.
- Modified the TskSearch interaction with the option to search using tags.
- Modified the EntXtsk and MemXtsk objects so that:
  - the getUsrRecno method returns 1 (system user) if the ownerType is not “U”,
  - the setUsrRecno method sets both the OwnerType (to “U”) and OwnerRecno properties,
  - new properties and corresponding get/set accessors for OwnerType and OwnerRecno, and
  - new property called OwnerName was added. OwnerName represents the GrpName if OwnerType is set to “G” and represents UsrLogin if OwnerType is “U.”

Web Services SDK

- Added support for the following interactions:
  - TagDelete removes a tag instance from the Master Data Engine.
  - TagPut creates or updates a tag instance in the Master Data Engine.
  - TagGet retrieves a tag from the Master Data Engine.
- TagReset resets the given tag types. All tag instances for the tag type are deleted and all tasks are reprocessed to determine if the tag applies to each task.
- TskPutByCriteria assigns the specified owner to every task in the system that matches the specified criteria (for example, assigns all Potential Duplicate tasks to User 1).
- TskSearchCount returns the count of every task in the system that matches the specified criteria and, optionally, the count of each tag type.
- GroupsUpdate creates or updates group DN information.
- GroupsSync is used to perform group synchronization with the LDAP server.
- The CmpSpcWs.java method was added to the cmpspec classes to support getting and setting the comparison mode (cmpmode).
- Added the methods getCmpMode (returns integer) and setCmpMode (accepts integer).
- Added getTags() methods.
- Added a new TskKey class to support input to TagGet.
- Added new TagGet bean.
- Modified the TskSearch interaction with the option to search using tags.
- Modified the EntXtsk and MemXtsk objects so that:
  - the getUsrRecno method returns 1 (system user) if the ownerType is not “U”,
  - the setUsrRecno method sets both the OwnerType (to “U”) and OwnerRecno properties,
  - new properties and corresponding get/set accessors for OwnerType and OwnerRecno, and
  - new property called OwnerName was added. OwnerName represents the GrpName if OwnerType is set to “G” and represents UsrLogin if OwnerType is “U”.
- Modified Web Service MemGet interaction to support multiple inputs, which matches the behavior of the .NET and Java APIs. This behavior is such that MemGet can handle one or more MemHeads. If more than one is specified, then a vector or list of members is returned. This enables the API caller to make a single request/reply instead of a separate request/reply for each member needed.

Version 8.1

.NET SDK

- RelTskGet, RelTskDelete, RelTskResolve, and RelTskSearch interactions have been added to support relationship task management.
- RelPathToRoot interaction supports hierarchy management and is used to retrieve a path from one node (entRecno at the starting point) to the root of the tree in the relationship.
- RelGet was renamed to RelSearch and modified to support other relationship retrieval methods. Any client applications written using version 8.0 will need to have RelGet changed to RelSearch in the code.
- RelCountGet renamed to RelSearchCount. Any client applications written using version 8.0 will need to have RelCountGet changed to RelSearchCount in the code.
- RelPut interaction was updated to support RelLinks with RelAttrBeans.
RelLink updated with collections of RelAttrBeans by attrCode.

RelAttrBeans is a dynamically created interaction with the appropriate fields based on the SegHead and RelSegAttr definition.

RelSegAttr is a new dictionary row bean for configuring relationship attributes (RelAttrs).

RelRule is a new dictionary row bean for configuring relationship linking rules.

RelXtsk is a new bean for all the IxnRelTsk interactions.

RelXtskCriteria is a new bean used as input criteria by RelTskSearch.

Added a new ixnStat dictionary segment added to track interaction history by hour.

Removed the following segments: bktType, excHead, excFunc, and excSpec.

GetArgs and SetArgs methods were added to the Ixn class.

The setUserPassEncrypt method is no longer supported in the APIs. The use of LDAP directory server requires that passwords be passed to the engine unencrypted. To address any security concerns, it is recommended that you run SSL between the client-Engine and Engine-LDAP.

Java SDK

- RelTskGet, RelTskDelete, RelTskResolve, and RelTskSearch, interactions have been added to support relationship task management.
- RelPathToRoot interaction supports hierarchy management and is used to retrieve a path from one node (entRecno at the starting point) to the root of the tree in the relationship.
- RelGet was renamed to RelSearch and modified to support other relationship retrieval methods. Any client applications written using version 8.0 will need to have RelGet changed to RelSearch in the code.
- RelCountGet renamed to RelSearchCount. Any client applications written using version 8.0 will need to have RelCountGet changed to RelSearchCount in the code.
- RelPut interaction was updated to support RelLinks with RelAttrBeans.
- RelLink updated with collections of RelAttrBeans by attrCode.
- RelAttrBeans is a dynamically created interaction with the appropriate fields based on the SegHead and RelSegAttr definition.
- RelSegAttr is a new dictionary row bean for configuring relationship attributes (RelAttrs).
- RelRule is a new dictionary row bean for configuring relationship linking rules.
- RelXtsk is a new bean for all the IxnRelTsk interactions.
- RelXtskCriteria is a new bean used as input criteria by RelTskSearch.
- Added string modification to meet predefined entity XML compliance standards in methods toXMLString and toSmallXMLString.
- Added a new ixnStat dictionary segment added to track interaction history by hour.
- Removed the following segments: bktType, excHead, excFunc, and excSpec.
- GetArgs and SetArgs methods were added to the Ixn class.
- In 6.x and earlier versions of the Java SDK, MemRow would automatically load the attribute name (attrName). In later versions, the toXMLString methods were made part of the Row class so that any row could be output as XML, including implementation-defined segments and dictionary rows. Not all rows consist of the same fields so the segment definition must be used to determine which
fields are included in the output. attrName and attrLabel are part of the dictionary, not part of member data segment definition, so they are not included in the output. Users who want functionality similar to the previous method will need to use the attrCode or attrRecno to identify which attribute is being output (both part of the member data segment definition). You can create a Dicstore and use one of three methods to get the attrName and attrLabel: getSegAttrByRecno(), getSegAttrByCode() or populateAttrNames(). You will need to format your own XML strings to include this data per your business requirements.

- The setUsrPassEncrypt method is no longer supported in the APIs. The use of LDAP directory server requires that passwords be passed to the engine unencrypted. To address any security concerns, it is recommended that you run SSL between the client-Engine and Engine-LDAP.

**Web services SDK**

- RelTskGet, RelTskDelete, RelTskResolve, and RelTskSearch, interactions have been added to support relationship task management
- RelPathToRoot interaction supports hierarchy management and is used to retrieve a path from one node (entRecno at the starting point) to the root of the tree in the relationship
- RelGet was renamed to RelSearch and modified to support other relationship retrieval methods. Any client applications written using version 8.0 will need to have RelGet changed to RelSearch in the code.
- RelCountGet renamed to RelSearchCount. Any client applications written using version 8.0 will need to have RelCountGet changed to RelSearchCount in the code.
- RelPut interaction was updated to support RelLinks with RelAttrBeans
- RelLink updated with collections of RelAttrBeans by attrCode
- RelAttrBeans is a dynamically created interaction with the appropriate fields based on the SegHead and RelSegAttr definition
- RelSegAttr is a new dictionary row bean for configuring relationship attributes (RelAttrs).
- RelRule is a new dictionary row bean for configuring relationship linking rules
- RelXtsk is a new bean for all the IxnRelTsk interactions
- RelXtskCriteria is a new bean used as input criteria by RelTskSearch
- Added a new ixnStat dictionary segment added to track interaction history by hour.
- Removed the following segments: bktType, excHead, excFunc, and excSpec.
- GetArgs and SetArgs methods were added to the Ixn class.
- The setUsrPassEncrypt method is no longer supported in the APIs. The use of LDAP directory server requires that passwords be passed to the engine unencrypted. To address any security concerns, it is recommended that you run SSL between the client-Engine and Engine-LDAP.

**C++ SDK**

The C++ SDK has been deprecated with this release.
Version 8.0

.NET SDK
- Added support for relationship management.
- Modified the way handlers are registered and deployed. Registration is now through IBM Initiate Workbench; deployment remains through madconfig. Refer to the Event Notification and Event Callout chapter in the SDK Training Guide for details.
- Add the MemScore interaction to enable scoring between records without requiring a Master Data Engine instance.

Java SDK
- Added support for relationship management.
- Modified the way handlers are registered and deployed. Registration is now through IBM Initiate Workbench; deployment remains through madconfig. Refer to the Event Notification and Event Callout chapter in the SDK Training Guide for details.
- Add the MemScore interaction to enable scoring between records without requiring a Master Data Engine instance.

Web services SDK
- Added support for relationship management.
- Add the MemScore interaction to enable scoring between records without requiring a Master Data Engine instance.

Version 7.5

.NET SDK
- Provides capabilities to develop callback handlers in support of engine callouts and event notifications.

Java SDK
- Modified to support 64-bit identifiers. All clients (including customer-developed) that consume the Java SDK will need to be recompiled and tested. Developers will need to pay special attention to use of "int" and "long" when dealing with modified key attributes (audrecno, entrecno, memrecno, setrecno, bkthash, bktval, bktrole, and seqnum). You will want to make sure that you do not down cast the "long" into an "int" type.
- Added capabilities to develop callback handlers in support of engine callouts and event notifications.

Web services SDK
- Modified to support 64-bit identifiers
**Version 7.2**

**Java SDK**
- Enhanced to provide SSL support.
- Updated for French Canadian translation.

**Web services SDK**
- Enhanced to provide SSL support.
- Updated for French Canadian translation.

**Version 7.0**

**Java SDK**
- IxnEiaGet KeyType default is now MEMRECNO, for consistency.
- Added new classes: StrCmap, BktXgen.
- Added Row(ctx,SegCode) constructor for engine/context specific creation.
- Added MemRow(ctx,SegCode) constructor for engine/context specific creation.
- Context now contains m_metaCtx to support MemRow/Row creation in a multi-engine environment.
- Added Context.createRow(segCode) for MemRow creation in a multi-engine environment.
- Added MetaContext and MetaBase.
- Moved the majority of the Meta class up to abstract MetaBase class.
- MetaData now takes an abstract MetaBase class.
- MpiNet constructor needs a MetaBase that is from the Context, it is then passed in to Connect().
- Connect() method now takes the MetaContext passed in and sets it up with its own copy of MetaData.
- doSendRecv() now also takes a Context, need to get Meta for its call to connect.
- FldDef.java support VarChar for implementation-defined segments.
- MpiNet.java calls to msg.getRowList() now send in required Context sensitive Meta.
- exMemGet uses MEMATTRALL on segCodeFilter.
- Added set/get for the following:
  - mpi_enttype.entTypeDesc
  - mpi_memtype.memTypeDesc
  - mpi_dvdhead.dvdDesc
  - mpi_dvdxstd.cmapStrCode
  - mpi_dvdxstd.stdRole1Label
  - mpi_dvdxstd.stdRole2Label
  - mpi_dvdxstd.dvdDesc
  - mpi_dvdxcmp.cmpRoleLabel
  - mpi_dvdxcmp.dvdDesc
  - mpi_dvdxqry.qryRoleLabel
  - mpi_dvdxqry.dvdDesc
  - mpi_dvdxbkt.bktGrpLabel
- mpi_dvdxbkt.dvdDesc
- mpi_dvdbybkt.dvdDesc
- mpi_cmphhead.dvddesc
- mpi_cmpspec.cmpDesc
- mpi_cvwhead.memptypeno
- mpi_stdfunc.stdtype1 and stdtype2
- mpi_srcxent.srcprior

- Removed set/get for the following:
  - mpi_stdfunc.minroles and maxroles
  - mpi_dvdbybkt.freqstrcode and maxbktfreq

- Added getDdlMsg() to MpiNet.java
- Added LinkType to MemHead, MemLink, and EntLink
- Metadata length has been updated to fifteen for state name in MemAddr,
  MemExta, and MemExtd
- Added minBktRoles and maxBktRole to EntType
- Removed dvdFuncCode from DvdXcmp, DvdXqry, and DvdXbkt
- Added isEnhanced (y/n) to CvwHead
- Added attrPrior to CvwXseg. If this field is set to 0 (zero), then the behavior
  remains as it has always been. For any given attribute, the record from source 1
  is of equal importance to source 2. However, if for any given set of cvwXseg
  records with the same attrrecno, the attrprior[ity] field implies that attribute
  records from different sources have a priority. attrprior 1 has a higher priority
  than attrprior 2, and so on.

  Note that more than one source can share the same attrprior level, in which case
  the maudrecno/timestamps come into play. You can mix the old behavior and
  the new behavior on a per attrrecno basis in the same composite view.

- exTskGet added MEMNAME to segCodeFilter.
- IxnMemSearch updated Javadoc HTML documentation for MaxRows() and new
  SegCodeFilter options.
- IxnMemGet updated Javadoc HTML documentation for new SegCodeFilter
  options: ALL (all segments), MEMALL (all member segments), DICALL (all
  dictionary 23

Web services SDK

- Web services now uses RPC/Literal binding for compatibility with .NET 2.0
- Updated all C# examples for use with .NET 2.0
- Everything that returned a Dictionary now returns a Dictionary[]
- Java Axis section of the HowToUseExamples.txt removed -W option as wrapped
  supported
- Added exSegDefGet.cs
- Changed exGetDictionary.cs to process DicPut Insert, DicGet, and then DicPut
  Delete.
- All Java examples updated to deal with Dictionary[] change (UsrInfoSearch,
- SegmentDefGet.java is no longer case sensitive.
- IdentityHub.java changed to returns Dictionary[].
- web.xml updated
• TaskSearchRequest, TaskGetRequest updated for new SegCodeFilter options – ALL and MEMALL.
• MemberSearchRequest updated for SegCodeFilter options ALL and MEMALL, and new external entity getTypes.
• Updated MaxRow java doc.
• MemberGetRequest updated for SegCodeFilter options ALL and MEMALL and new entity getTypes.
• cmpDesc added to CmpSpecWs and CmpHeadWs.
• linkType added to EntLinkWs and MemLinkWs.
• dvdDesc and qryRoleLabel added to DvdXqryWs; removed dvdFuncCode.
• dvdDesc added to DvdHead.
• stdRole1Label, stdRole2Label, cmapStrCode, and dvdDesc added to DvdXStdWs.
• dvdDesc and dvdGrpLabel added to DvdXbktWs; removed dvdFuncCode and FreqStrCode.
• dvdDesc added to DvdYbktWs; removed FreqStrCode and maxBktFreq.
• dvdDesc and cmpRoleLabel added to DvdxcmpWs; removed dvdFuncCode.
• stdType1 and 2 added to StdFuncWs; removed minRoles and maxRoles.
• EntTypeDesc, minBktRole, and maxBktRole added to EntTypeWs.
• linktype added to MemHeadWs.
• attrPrior added to CvwXsegWs.
• memTypeno and isEnhanced added to CvwHeadWs.
• Dictionary.java added new tables:
  – StrType
  – WgtType
  – StdType
  – BktType
  – StrCmap
  – BktXgen
• Removed DvdFunc; does Null check for empty array.
• Updated Java API JAR.
• Updated the web services SDK to use Axis 1.4.
• Enhanced the web services SDK to honor entType and entTypeNo
Appendix B. Member put interaction—options and behavior

In order to understand some of the concepts in this document, some background information is needed. The Master Data Engine operates on a set of data values that are known as a member. A member can represent anything—a customer record, an inventory part record, or a hotel guest. A member object is made up of a header record which uniquely identifies the member and zero or more attributes. Attributes are the data values, which describe the member. An attribute code defines a value of a particular type. It is legal to have multiple values for the same attribute code. For instance, many customers may have more than one PHONE value.

For the purpose of the explanations below, assume an IBM Initiate Master Data Service implementation where the attributes in Table 20 make up a complete customer member.

Table 20. Example Customer Member Attributes

<table>
<thead>
<tr>
<th>Attribute Code</th>
<th>Example Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAME</td>
<td>John Q. Public</td>
</tr>
<tr>
<td>ADDR</td>
<td>1043 W. Easy Street, Phoenix AZ</td>
</tr>
<tr>
<td>PHONE</td>
<td>602-555-1212</td>
</tr>
<tr>
<td>CUSTID</td>
<td>120389</td>
</tr>
<tr>
<td>TAXID</td>
<td>AZ-30928393</td>
</tr>
</tbody>
</table>

The Master Data Engine has the ability to store more than one value for each type of attribute in a Hub. For instance, a customer might have more than one phone, so there needs to be the ability to represent more than one phone number. The Master Data Engine also has the ability to store historical (inactive) values for a given attribute. When a customer moves from one address to another, the old address becomes inactive, while the new value becomes the active value.

To support these concepts, each member attribute segment in the system contains a record status value, (abbreviated as RecStat), which shows the status of a particular attribute value. There are currently four valid record status values: Active, Inactive, Shadow, and Deleted.

Another concept required for a good understanding of the member put options is that of the row indicator or rowInd. The row indicator is a field value that is not stored in the database, but is used to indicate how the attribute value is to be stored. Row indicator values can be: Insert, Update, Delete, and Ignore. Row indicators are only honored by explicit puts, which are described later in the document.

The IxnMemPut and IxnMemPutBulk interactions now support the change request (CR) functionality used by the Data Trust application.

**Important:** If CRs are enabled and one of these interactions triggers a CR based on the configuration, the result of the interaction will be an empty {} instead of the data that was modified.
Overview

The member put interaction has several options that control how member data is stored in the database. Understanding these options, and how they interact with various system configuration settings, is key to getting data stored in the form you want it. This appendix covers the difference between implicit and explicit puts, the various matching modes and their behavior in various system configurations, and the put type options that help you limit what kind of updates can take place.

The descriptions of these options are not programming language specific to any of the APIs supported by Initiate Systems. If you need language specific help on the syntax of these options, refer to any of the API references.

Implicit put modes

There are two main styles, or modes, in which puts can occur: implicit and explicit. These put variations are controlled by the Member mode option. There are three variations that are considered to be implicit modes. Implicit mode is defined as when the put interaction passes data to the Master Data Engine, and allows it to insert, update, and change the record status of the member rows based on what is contained in the input, and what data (if any) is already contained in the Master Data Engine database. When using these modes, the engine applies the input data in the incoming interaction to the data contained in the database based on various configuration rules, and the specific implicit member mode that was used. When using an implicit mode, row indicator values are not used to determine the state of the row. Record status values that come as part of the input are not honored. The newly inserted rows get member sequence values assigned automatically. The engine examines the incoming data, and bases the updated row storage on the configuration, and the row logic dictated by the implicit member mode.

Note: To logically delete an attribute when using an IMPLICIT type, you can create an empty member attribute row with the AttrRecno or AttrCode and set the RowInd to D to delete the most recent occurrence of that attribute.

The three types of implicit member modes are: PARTIAL, ATTRCOMP and COMPLETE. They are explained in detail below.

PARTIAL

This option is used when a source system sends an update to a member, but you do not know if this is a complete picture of member, or if you have the complete range of values for a given attribute. For example, a given source system may send in an update for a member that contains just a value for the PHONE attribute, and does not send any other values. The PARTIAL option tells the system not to make any record status adjustment to any of the attributes. Because the example update just contained a value for phone, this does not mean that judgments should be made about the status of the NAME attribute. Use PARTIAL to denote that there is not any knowledge if this update represents the complete universe of values that represent this member. The engine will not make changes in status to the attributes, as it does not have a complete picture of the values for any of the attributes present in the interaction.

The results of a PARTIAL put depend on the values that are sent in, and the values of the nsexists and nsactive configuration settings which are described in more detail below. To simplify the explanation, assume that nsexists is set to zero, which allows an unlimited number of attribute values for each attribute code.
If nsactive is equal to zero: All new and updated attribute values will get a record status of Active.

If nsactive is greater than zero (assume nsactive = 2 for this example): The latest (last updated) two attribute values would receive an Active status. Any other active attribute values would have their record status set to Inactive (historical).

**ATTRCOMP**

This mode stands for attribute complete. Like the PARTIAL mode, the ATTRCOMP mode tells the software that it may not have a complete picture of all the attributes that make a complete member, but for the attributes that are present, all known values for the member are included in the member put interaction.

ATTRCOMP enables Potential Overlay processing. A Potential Overlay condition occurs when two sets of disparate attribute values are applied to the same member (as identified by a given source code and member ID number). When the Master Data Engine sees this mode setting, it compares the attribute set in the input to an image of the member from the database, and if that comparison score is equal to or less than the overlay threshold, a Potential Overlay task is created (if the system has been configured to support this task type).

Using the example attribute set above, a given interaction with a member mode set to ATTRCOMP, and passing in NAME and PHONE, tells the engine that this is the complete range of values for these two attributes, and that adjustments should be made to the record status based on the configuration options that control record status (see nsactive below).

- If nsactive is equal to zero: Each attribute value that is present in the put interaction would have a record status of Active; any previously active values not present in the input would be set to Inactive. If one value were sent in, one would be active. If two were sent in, two would be set to active.
- If nsactive is greater than zero (assume nsactive = 2 for this example): The latest (last updated) two attribute values would receive an Active status. Any other active attribute values would have their record status set to Inactive (historical).

**COMPLETE**

This mode tells the engine that the input to the member put interaction contains all of the values, for all of the attributes defined for this member type. As far as record status updates, the COMPLETE mode works exactly like ATTRCOMP. However, in the COMPLETE mode if partial attributes are sent for an update, the RecStat of those attributes which are already present in the database—but not received in this update—are changed to ‘I.’

COMPLETE enables Potential Overlay processing. A Potential Overlay condition occurs when two sets of disparate attribute values are applied to the same member (as identified by a given source code and member ID number). When the Master Data Engine sees this mode setting, it compares the attribute set in the input to an image of the member from the database, and if that comparison score is equal to or less than the overlay threshold, a Potential Overlay task is created (if the system has been configured to support this task type).
Explicit put mode

In situations where a developer wants to control exactly what is stored, and the record status of the attributes being stored, a member mode of EXPLICIT should be used. The EXPLICIT mode does not apply any logic as to which attributes should be updated, inserted or deleted. It does not vary the status based on the configuration. You must set the row indicator and record status of each and every attribute that is being updated. For example, if you have an existing member that you want to update, you must read in that member via a member get interaction, set the row indicator value for each row to UPDATE and set the record status field to the value you want reflected in the stored data.

The Master Data Engine determines the action to be taken on the member row objects passed in, based on the value of their RowInd attribute. Use the setRowInd() function to set the attribute to one of the following values.

- **RowInd.IGNORE** – No action is taken
- **RowInd.INSERT** – Member data is inserted into the Master Data Engine database
- **RowInd.UPDATE** – Member data is updated in place
- **RowInd.DELETE** – Member data is physically deleted from the Master Data Engine database

Explicit puts also check a value that is stored in the memHead record called memVerno. This value is used as a version tag that aids the Master Data Engine to reduce lock contention by performing optimistic concurrency. If you retrieve a member with a memVerno = 1 and send in an explicit put after another process has already updated the member, you will receive a error message stating that the member you are trying to update is out of date. Implicit puts do not use versioning, as they are not concerned with the before picture of the data. The input of an implicit put is merely applied via the rules described above.

Explicit mode puts can be used to create attributes with a ‘Shadow record status, or a logically ‘Deleted status. You can also use explicit puts to physically delete an attribute by using the DELETE row indicator value.

**Note:** It is good practice to set RowInd to INSERT, which is the default behavior.

Put type options

The member put interaction provides the ability to restrict how members are added or updated in the system. There are three valid options for the Put Type setting, as shown in Table 21.

<table>
<thead>
<tr>
<th>Option</th>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>INSERT_ONLY</td>
<td>Restricts the Master Data Engine to creating a new member. If a member already exists for this srcCode/memIdnum, the interaction will fail with an error code of EEXISTS.</td>
</tr>
<tr>
<td>UPDATE_ONLY</td>
<td>Restricts the Engine to updating existing members only. If an attempt is made to update a member that does not already exist, the interaction will fail with an error code of ENOREC.</td>
</tr>
<tr>
<td>INSERT_UPDATE</td>
<td>Adds a member if one does not exist, or it will update the existing member if it does exist</td>
</tr>
</tbody>
</table>
These options are useful if you are writing programs that want to just add new data without having to do a read/get operation to find out if the member exists, or to update members that exist without worrying about adding any new members. The INSERT_UPDATE option can be used if you need a combination of the two behaviors.

**Note:** This option works at a member level, not an attribute level.

### Match mode options

The member put interaction works in two phases. First, it updates the member data in the database based on the input, and the other options provided to the interaction. Next, it cross matches the member against any similar members to see if they should be linked, or if a task should be created for manual review. In previous versions of the Initiate® software, the match mode options would control whether or not this behavior should take place. In versions 5.0 and later, the matching is controlled by the configuration settings haslink and hasxtsk, as described below. The match mode options are still in place to provide compatibility with older code, but they have no effect on the matching behavior.

The match mode options are defined in the Table 22.

**Note:** The following behaviors hold relevance when the Entity Manager is in asynchronous mode (with the exception of DONOTHING, which operates in both synchronous and asynchronous mode). However, when the Entity Manager is in synchronous mode, members are matched immediately, that is, the option does nothing.

*Table 22. MemPut Match Mode Options*

<table>
<thead>
<tr>
<th>Option</th>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>DONOTHING</td>
<td>Update the member data, but do not cross match the updated data against other members in the Hub database. The interaction returns control to the caller as soon as the update is complete. This blocks this member from being matched when it is put, but another put that has buckets in common could cause this member to be matched.</td>
</tr>
<tr>
<td>DEFERRED</td>
<td>Member matches when Entity Manager gets to this spot in the queue</td>
</tr>
<tr>
<td>IMMEDIATE</td>
<td>Cross match the member immediately following the update of the member data. This option has no additional effect in asynchronous mode.</td>
</tr>
</tbody>
</table>

### Configuration

The following configuration settings have an effect on the member put options described above. This is not a complete listing of configuration items. The items described in Table 23 affect the behavior of the member put options described earlier in this document.

*Table 23. Configuration Settings*

<table>
<thead>
<tr>
<th>Table</th>
<th>Column</th>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>mpi_segattr</td>
<td>nsactive</td>
<td>Controls the number of simultaneously active attributes. This setting determines how many attributes retain an Active record status, while other (previously active) attributes are updated to an Inactive (historical status).</td>
</tr>
</tbody>
</table>
Table 23. Configuration Settings  (continued)

<table>
<thead>
<tr>
<th>Table</th>
<th>Column</th>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>mpi_segattr</td>
<td>nsexists</td>
<td>Control the number of simultaneously existing attributes during implicit puts. If the number of attributes stored exceeds this value, the oldest attributes will be physically deleted as part of a self-pruning operation.</td>
</tr>
<tr>
<td>mpi_enttype</td>
<td>isasync</td>
<td>Controls whether or not cross match operations for this entity type are synchronous or asynchronous. Synchronous operations cross match the member against similar members before the interaction returns. Asynchronous matching returns control to the interaction caller as soon as the member data is updated. Matching occurs in the background.</td>
</tr>
<tr>
<td>mpi_srched</td>
<td>pocmpcode</td>
<td>Specifies the comparison code (if any) used to determine if a member update will result in a Potential Overlay task. This comparison routine is used when an incoming member put interaction has a member mode which is set to COMPLETE or ATTRCOMP.</td>
</tr>
<tr>
<td>mpi_srched</td>
<td>poscore</td>
<td>This is the threshold used in Potential Overlay processing.</td>
</tr>
<tr>
<td>mpi_enttype</td>
<td>haslink</td>
<td>Controls whether or not entity linking should occur.</td>
</tr>
<tr>
<td>mpi_enttype</td>
<td>hasxtsk</td>
<td>Controls whether Potential Linkages and other member states should be captured in the form of a workflow task.</td>
</tr>
</tbody>
</table>

### Entity management priority

In IBM Initiate Workbench, you can specify a Default Entity Priority for each definitional source. Another option for entity management priority is to set an entity management priority at the time of the member write. This enables you to set a lower priority for batch loaded members regardless of the associated source.

To accomplish this, a new function has been implemented for the member put interactions (MemPut, MemPutBulk, and MemPutQual). When the setEntPrior function is used, the entity manager honors the priority, and processes the specific member record first. If disabled, then entity management defaults to processing records in caudRecno order or source entity management priority. The set priority of a member will override the default entity management priority (e.g., the source entity management priority).

- setEntPrior – Sets the entity management priority. All members in a member put interaction (MemPut, MemPutBulk, MemPutQual) will have this priority rather than the source default. The value is set in the mpi_entique.wrkprior field.

The default setting for the entity management priority (EntPrior) is 0 and instructs the entity manager to use the source’s Default Entity Priority setting.
Appendix C. Error codes

This section contains error codes and their descriptions. A complete listing of numeric error codes can be found in `hub_install_path/conf/ hubmessages_en_US.properties`.

Table 24. Error Codes

<table>
<thead>
<tr>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OK</td>
<td>No error.</td>
</tr>
<tr>
<td>EPERM</td>
<td>Invalid permissions.</td>
</tr>
<tr>
<td>ECOMM</td>
<td>Communications error — lost server connection.</td>
</tr>
<tr>
<td>EODBC</td>
<td>ODBC error — reported from the ODBC driver or database directly.</td>
</tr>
<tr>
<td>ETRUNC</td>
<td>Reporting that a string was stored which was longer than the defined field into which it was inserted.</td>
</tr>
<tr>
<td>ENOTCONN</td>
<td>Master Data Engine server is not connected.</td>
</tr>
<tr>
<td>EINVAL</td>
<td>Invalid values provided in incoming parameters or datatype, including multiple parent values and circular references in a hierarchical path.</td>
</tr>
<tr>
<td>ENOMEM</td>
<td>Out of memory error — no working memory could be allocated.</td>
</tr>
<tr>
<td>ENOREC</td>
<td>No record found in the database.</td>
</tr>
<tr>
<td>ENOLIB</td>
<td>Runtime loadable libraries were not found.</td>
</tr>
<tr>
<td>ENOFUNC</td>
<td>Pointed-to function was not available.</td>
</tr>
<tr>
<td>ELOCKED</td>
<td>A record cannot be updated because it is locked.</td>
</tr>
<tr>
<td>EEEXISTS</td>
<td>A record insert is requested, but the record already exists.</td>
</tr>
<tr>
<td>EFILEIO</td>
<td>File I/O error — could not read or write from a file.</td>
</tr>
<tr>
<td>EVERSION</td>
<td>Version mismatch, caused by opportunistic locking.</td>
</tr>
<tr>
<td>ESTATUS</td>
<td>The requested attribute status does not exist.</td>
</tr>
<tr>
<td></td>
<td>An attempt was made to add or update a logically deleted member.</td>
</tr>
<tr>
<td>EACTIVE</td>
<td>Not in use.</td>
</tr>
<tr>
<td>EINACTIVE</td>
<td>Not in use.</td>
</tr>
<tr>
<td>EDELETED</td>
<td>An attempt was made to add or update a logically deleted member.</td>
</tr>
<tr>
<td>ESHADOW</td>
<td>Not in use.</td>
</tr>
<tr>
<td>EOBsolete</td>
<td>An attempt was made to apply a change or update to an obsolete record.</td>
</tr>
<tr>
<td>EINTEGRITY</td>
<td>A data integrity issue; i.e., violation of a unique constraint.</td>
</tr>
<tr>
<td>EDISABLED</td>
<td>A user ID had been made inactive (or disabled).</td>
</tr>
<tr>
<td>EINSANE</td>
<td>The system has detected a condition or state which it believes should not be possible and does not have the information to explain how it got to this state.</td>
</tr>
<tr>
<td>EPANIC</td>
<td>The system has detected a critical error or condition. The system should be shut down to prevent further data inconsistencies.</td>
</tr>
<tr>
<td>ENUNKNOWN</td>
<td>An unknown error.</td>
</tr>
</tbody>
</table>
Legal Statement

Licensed Materials – Property of IBM

- Use, duplication or disclosure restricted by GSA ADP Schedule Contract with
IBM Corp. IBM, the IBM logo, InfoSphere, Initiate, and Initiate Master Data Service
are trademarks of IBM Corp., registered in many jurisdictions worldwide. Java and
all Java-based trademarks and logos are trademarks or registered trademarks of
Oracle and/or its affiliates. Other product and service names might be trademarks
of IBM, or other companies. This Program is licensed under the terms of the
license agreement accompanying the Program. This license agreement may be
either located in a Program directory folder or library identified as "License" or
"Non-IBM License", if applicable, or provided as a printed license agreement.
Please read this agreement carefully before using the Program. By using the
Program, you agree to these terms.
Notices and trademarks

This information was developed for products and services offered in the U.S.A.

Notices

IBM may 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 described in this document. The furnishing of this document does not grant 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 character set (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan Ltd.
1623-14, Shimotsuruma, Yamato-shi
Kanagawa 242-8502 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 publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication 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
J46A/G4
555 Bailey Avenue
San Jose, CA 95141-1003 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 measurements 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.

All statements regarding IBM’s future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information is for planning purposes only. The information herein is subject to change before the products described become available.

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. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

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**

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide.

Other product and service names might be trademarks of IBM or other companies.


The following terms are trademarks or registered trademarks of other companies:

Adobe is a registered trademark of Adobe Systems Incorporated in the United States, and/or other countries.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Microsoft, Windows, and Windows NT are trademarks of Microsoft Corporation in the United States, other countries, or both.

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

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.
Index

A
accessing beans in handlers 112
adding rows to RowLists 41
ATTRCOMP put mode 131
audit get interaction 85
audit history search interaction 85
autolink threshold 15

B
bucketing 13

C
CallbackHandlerException 114
callout handlers
accessing beans 112
error handling 114
logging messages 113
clerical review threshold 16
clients 6
compare
about 15
data 13
comparing member interactions 64
configuration information 71
comparison process 15
compatibility with earlier SDK versions ix
compiling Java handler class files 114
COMPLETE put mode 131
components and applications of Initiate Master Data Service 4
configuration settings, member put options 133
connections
establishing 29
MPINET for Java SDK 29
MPINET over SSL for Java SDK 29
creating custom handlers 105
creating implementation-defined segments
Java 44
web services 45
customer support contacting 145

d
data model
web services 27
debugging enabling 27
DEFERRED match mode option 133
derivation 12
dictionary class (web services SDK) 51
dictionary data
about 21
dictionary data (continued)
group head 21
segment attributes 22
segment head 21
source head 21
user head 21
dictionary get interaction 80
dictionary interactions 80
dictionary put interaction 81
dictionary segments 33
DONOTHING match mode option 133
dictionary get interaction 94
dictionary interactions 92
dictionary put interaction 95
dictionary search interaction 93
entity management priority 134
entity types
definition 9
enumerated data types 22
derror codes
EACTIVE 135
ECOMM 135
EDELETED 135
EDISABLED 135
EXISTSS 135
EFILEIO 135
EINACTIVE 135
EINSANE 135
EINTEGRITY 135
EINVAL 135
ELOCKED 135
ENOBUFNC 135
ENOLIB 135
ENOMEM 135
ENOREC 135
ENOTCONN 135
ENUNKNOWN 135
EBOSOLEITE 135
EODBC 135
EPANIC 135
EPERM 135
ESHADOW 135
ESTATUS 135
ETRUNC 135
EVERSION 135
OK 135
error handling 114
establishing connections 29
event handlers deploying 115
examples
about
SDK Reference 56
handler
Java 106, 108
writing event information 106
Java
IxnDicGet 80
examples (continued)
Java (continued)
IxnDicPut 81
IxnEiaGet 94
IxnEiaPut 95
IxnEiaSearch 93
IxnTskGet 88
IxnTskPut 89
IxnTskSearch 86
IxnUsrSetPass 84
memComp 66
memDelete 74
memDrop 76
memGet 60
memMatch 63
memMerge 72
memScore 77
memSearch 57
memUnmerge 73
record matching
SDK Reference 7
web services
deleteMember 74
dropMember 76
getDictionary 80
getEia 95
getMember 61
matchMember 64
Member class 49
memberScore 78
mergeMember 72
putDictionary 82
putEia 96
putTask 91
putUserPassword 84
searchEia 93
searchMember 59
searchTask 87
undeleteMember 75
unmergeMember 73
explicit PUT mode 132
get relationship history interaction 99
get user info interaction 84
getting data from implementation-defined segments (Java) 46
going data from implementation-defined segments (web services) 47
getUserErrCode interaction 101
group get info interaction 101
group head 21
groups sync interaction 101
groups update interaction 100

© Copyright IBM Corp. 1995, 2011
H

handlers
accessing beans 112
compiling Java handler class files 114
creating custom 105
event 115
getErrText 101
interactions 101
Java callout 105
testing 115
hubs 5

I

identifiers 8
IDS (implementation-defined segments)
creating Java 44
getting data from Java 46
retrieving Java 48
IMMEDIATE match mode option 133
implicit PUT mode 130
implicit PUT modes 130
inbound message processing 7
Initiate Master Data Service components and applications 4
INSERT_ONLY put type option 132
INSERT_UPDATE put type option 132
installation 28
installing the SDK Java 28
interactions
audit get 85
audit history search 85
basic 55
comparing 64
dictionary get 80
dictionary put 81
EIA get 94
EIA put 95
EIA search 93
get user info 84
getUsrErrCode 101
group get info 101
groups sync 101
handler 101
IxnRelDelete 98
IxnRelGet 97
IxnRelGetHist 99
IxnRelPathToRoot 98
IxnRelPut 97
IxnRelSearch 97
IxnRelSearchCount 98
IxnRelTskDelete 99
IxnRelTskGet 99
IxnRelTskResolve 99
IxnRelTskSearch 99

interactions (continued)
member delete 74
member drop 76
member get 60
member get history 79
member match 62
member merge 72
member put 65
member score 76
member search 57
member undelete 75
member undo 75
member unmerge 73
member unput 65
memPut 130
qualified member put 65
relationship move 98
set user password 84
setArgs 101
tag delete 100
tag get 100
tag put 100
tag reset 100
task get 88
task get 2 89
task put 89
task put 2 92
task put by criteria 92
task search 86
task search count 92
workflow of 31
Ixn.getUsrErrCode 101
Ixn.setArgs 101
IxnAudGet 85
IxnAudSearchHist 85
IxnDicGet 80
IxnDicPut 81
IxnEiaGet 94
IxnEiaPut 95
IxnEiaSearch 93
IxnGroupGetInfo 101
IxnGroupsSync 101
IxnGroupsUpdate 100
IxnMemCompare 66
IxnMemDelete 74
IxnMemDrop 76
IxnMemGet 60
IxnMemGetHist 79
IxnMemMatch 62
IxnMemMerge 72
IxnMemPut 65
IxnMemPutQual 65
IxnMemScore 76
IxnMemSearch 77
IxnMemUndelete 75
IxnMemUndo 75
IxnMemUnmerge 73
IxnMemUnput 65
IxnRelDelete 98
IxnRelGet 97
IxnRelGetHist 99
IxnRelPathToRoot 98
IxnRelPut 97
IxnRelSearch 97
IxnRelSearchCount 98
IxnRelTskDelete 99
IxnRelTskGet 99
IxnRelTskResolve 99
IxnRelTskSearch 99

J

java handlers
JMX monitoring 115
Java SDK
MPINET connections 29
MPINET over SSL connections 29
Java SDK MPINET connections 31
JMX monitoring 115

L

legal notices 139
link
about 17
logging enabling 27
Java 113
logging handler messages 113

M

managed task queues 17
master data engine
process overview 10
Master Data Management 1
MATCH MODE options 133
matching
process 13
select 14
member compare interaction 66
MEMBER concept 129
member data
constructing 11
member delete interaction 74
member drop interaction 76
member get history interaction 79
member get interaction 60
member match interaction 62
member merge interaction 72
member objects 8
member put interaction 65
member score interaction 76
member search interaction 76
member search count 92
member segments 38
member undelete interaction 75
member unmerge interaction 73
MemName segment 32
memPut
overview 130
Contacting IBM

You can contact IBM for customer support, software services, product information, and general information. You also can provide feedback to IBM about products and documentation.

The following table lists resources for customer support, software services, training, and product and solutions information.

Table 25. IBM resources

<table>
<thead>
<tr>
<th>Resource</th>
<th>Description and location</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBM Support Portal</td>
<td>You can customize support information by choosing the products and the topics that interest you at <a href="http://www.ibm.com/support/entry/portal/Overview/Software/Information_Management/IBMInitiate_Master_Data_Service">www.ibm.com/support/entry/portal/Overview/Software/Information_Management/IBMInitiate_Master_Data_Service</a></td>
</tr>
<tr>
<td>Software services</td>
<td>You can find information about software, IT, and business consulting services, on the solutions site at <a href="http://www.ibm.com/businesssolutions/">www.ibm.com/businesssolutions/</a></td>
</tr>
<tr>
<td>My IBM</td>
<td>You can manage links to IBM web sites and information that meet your specific technical support needs by creating an account on the My IBM site at <a href="http://www.ibm.com/account/">www.ibm.com/account/</a></td>
</tr>
<tr>
<td>Training and certification</td>
<td>You can learn about technical training and education services designed for individuals, companies, and public organizations to acquire, maintain, and optimize their IT skills at <a href="http://www.ibm.com/software/sw-training/">http://www.ibm.com/software/sw-training/</a></td>
</tr>
</tbody>
</table>

Providing feedback

The following table describes how to provide feedback to IBM about products and product documentation.

Table 26. Providing feedback to IBM

<table>
<thead>
<tr>
<th>Type of feedback</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Product feedback</td>
<td>You can provide general product feedback through the Consumability Survey at <a href="http://www.ibm.com/software/data/info/consumability-survey">www.ibm.com/software/data/info/consumability-survey</a></td>
</tr>
</tbody>
</table>
Table 26. Providing feedback to IBM (continued)

<table>
<thead>
<tr>
<th>Type of feedback</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Documentation feedback</td>
<td>To comment on the information center, click the Feedback link on the top right side of any topic in the information center. You can also send comments about PDF file books, the information center, or any other documentation in the following ways:</td>
</tr>
<tr>
<td></td>
<td>• Online reader comment form: <a href="http://www.ibm.com/software/data/rcf/">www.ibm.com/software/data/rcf/</a></td>
</tr>
<tr>
<td></td>
<td>• E-mail: <a href="mailto:comments@us.ibm.com">comments@us.ibm.com</a></td>
</tr>
</tbody>
</table>