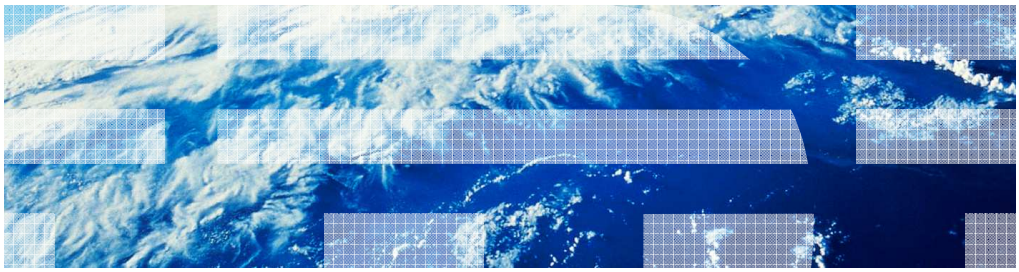
IBM WebSphere eXtreme Scale V8.6

eXtreme Data Format



This presentation describes support for the eXtreme Data Format feature included in IBM WebSphere® eXtreme Scale version 8.6.

eXtreme data format (XDF) overview

- XDF is the serialization capability and wire format used by the eXtreme IO(XIO) communication protocol when using COPY_TO_BYTES maps
- XDF is required in order to support the native .NET provider introduced in WebSphere eXtreme Scale 8.6
- XDF provides the following functional benefits :
  - Java and .NET applications can share the same cached data in an optimized compact format
  - Ability to view a string representation of data in the cache from the monitoring console without requiring the user classes on the servers
  - provides class evolution capabilities so that multiple versions of an application (or different applications) with different but compatible class definitions can share data within the grid.
  - Classes do not need to be defined as Serializable in order for XDF to process the class, so you can use XDF without needing to change your object definitions. XDF will serialize all non-static and non-transient fields for a given class (Note that for C# transient fields are determined by the use of NonSerialized attribute).

XDF refers to the serialization capabilities and the wire format that is used by eXtreme Scale 8.6 when using eXtreme IO as the communication protocol and using maps with the copy mode set to COPY_TO_BYTES

XDF is required in order to support the native .NET provider introduced in WebSphere eXtreme Scale V8.6.

XDF provides several functional benefits:

A common serialization format allows .NET and Java applications to share the same data in an optimized compact format.

A new mechanism identifies the fields that will participate in the partitioning calculation by using annotations in Java and attributes in C#.

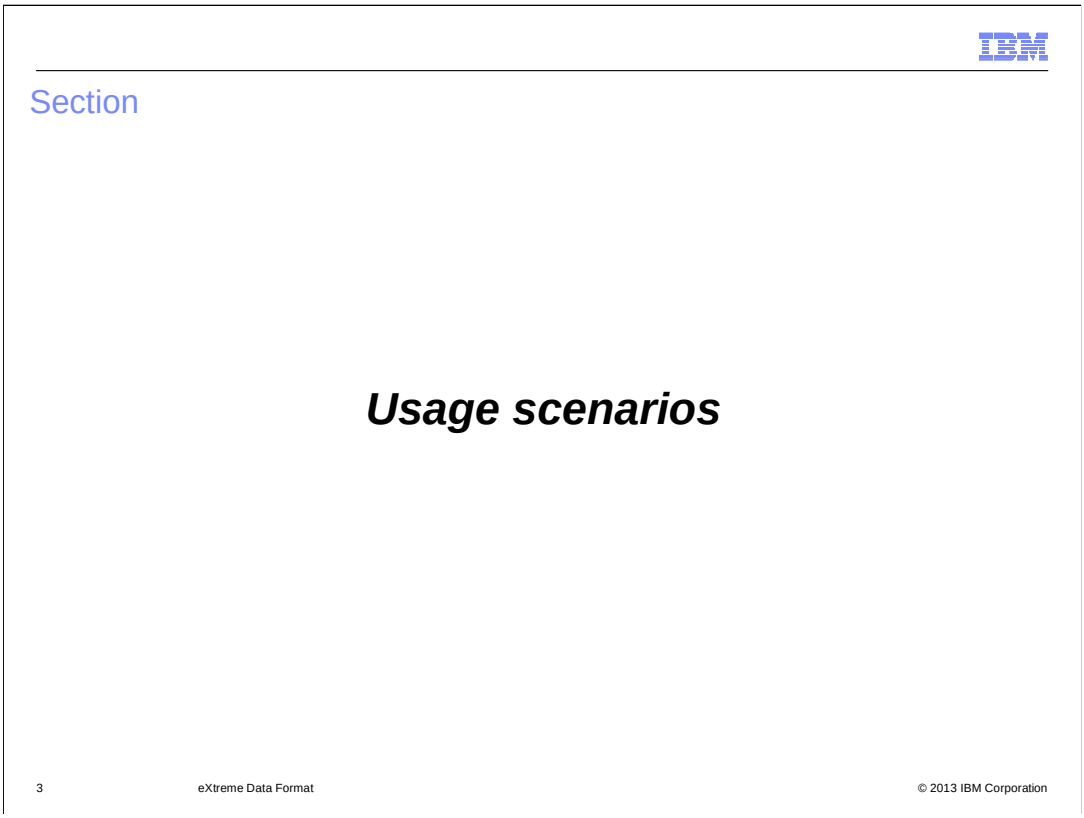
XDF introduces annotations for Java and attributes for C# to identify mappings between equivalent classes and fields that have different names but compatible types.

XDF provides you with the ability to include a string representation of the data within the monitoring console without requiring the user classes to be present on the container servers or the monitoring console. This feature is only available for maps that are using XDF.

XDF provides class evolution capabilities so that multiple versions of an application (or different applications) with different but compatible class definitions can share data within the grid.

Classes do not need to be defined as Serializable in order for XDF to process the class, so you can use XDF without needing to change your object definitions. XDF will serialize all non-static and non-transient fields for a given class. Note that for the C# programming language, transient fields are determined by the use of the NonSerialized attribute.

Additional features such as the OutputFormat configuration, runtime option, and the PluginOutputFormat annotation have been added to provide plug-ins and applications with more control over the serialization process.



Here are some usage scenarios of some of the XDF features for developing Java and C# enterprise applications that share data in a grid.

Type compatibility between Java and C#

- Language neutral data format
- Complete table in the documentation:
  http://pic.dhe.ibm.com/infocenter/wxsinfo/v8r6/topic/com.ibm.websphere.extremescale.doc/rxsxdfequiv.html
- Some of the types may have multiple compatible types for added flexibility
- Sample of some of the type mappings that are possible with XDF:

Java type	C# type
nt	int, uint, ushort
ava.lang.Integer	int?, uint?
ava.lang.String	String
ava.util.ArrayList	System.Collections.ArrayList, System.Collections.Generic.List
ava.util.HashMap	System.Collections.Generic.Dictionary, System.Collections.Hashtable
ava.util.Date	System.DateTime

One of the benefits of XDF is the language-neutral data encoding that allows Java and C# applications to share data in the same grid.

A subset of the compatible types is shown here. The link is to the complete table in the information center.

Some of the Java types – collections in particular – have multiple corresponding C# types that are considered to be compatible for the purpose of assignment. This lets you take advantage of generic collection types in C#.

Signed and unsigned integer values are also compatible – unsigned values that exceed the range of the signed values will be promoted to the next larger integral type if possible.

ClassAlias and FieldAlias

- By default for user classes to be compatible, fully qualified class names, field names and field types must match.
- ClassAlias and FieldAlias annotations provide the ability to match classes and fields with different names but compatible types.
- Example shows how to use the C# attribute to match an existing Java class

```java
package com.mycompany.department;

public class Employee {
    int empId;
    String name;
}
```

```csharp
namespace Com.MyCompany
{
    [ClassAlias("com.mycompany.department.Employee")]
    class Employee
    {
        [FieldAlias("empId")]
        int identifier;

        string name;
    }
}
```

5 eXtreme Data Format

One of the challenges when trying to share grid data across programming languages is that teams may already have existing naming conventions and packages or namespaces for their applications.

By default, in order to share data between C# and Java, the fully qualified class names, field names and field types must match.

The ClassAlias and FieldAlias annotations let you match classes and fields that do not have the same names, but do share common types. This feature will not enable you to share incompatible types. For example, a field named “ID” cannot be defined as a string in C# and as an integer in Java.

The example here illustrates how you could match a Java class on the left, with a C# class on the right that has a different qualified class name and a different field name. Notice that you only need to provide the annotation in one language (either C# or Java), provided that the aliases reference the class and field names from the other language. Also note that the “name” field does not require an attribute since it has the same name in both languages.

PartitionKey annotation and attribute

- This new annotation provides the application developer a way to indicate which fields in a class will contribute to the partitioning calculation – by default with XDF, all of the fields in the key will contribute to the partition calculation. XDF serializes the fields of the key and calculates a hashCode on the serialized bytes that is used for the partitioning scheme.
- The order attribute for the annotation defines the order in which the fields are processed while calculating the hashCode used for partitioning (since the order is important).

Example 1: Partition EmployeeKey and DepartmentKey based on deptId using field annotations

```
public class EmployeeKey {
    int empId;

    @PartitionKey
    int deptId;
}


public class DepartmentKey {
    @PartitionKey
    int deptId;

    int countryId;
}
```

Example 2 : Partition employee based on a nested reference to a field in the DepartmentKey class

```
public class EmployeeKey {
    int empId;

    @PartitionKey( value="deptId", order=0)
    DepartmentKey deptKey;
}


public class DepartmentKey {
    @PartitionKey(order=0)
    int deptId;

    int countryId;
}
```

6 eXtreme Data Format

The partitioning feature of eXtreme Scale is one of the mechanisms that controls the distribution of data across the container servers.

By default, all the fields in a key object will contribute to the partition calculation. In many cases this is an acceptable default, and these annotations do not need to be used.

However, in situations where you want to have some control over the placement of the data, then these annotations give you control over which fields will contribute to the partitioning calculation.

For example, assume that you want to co-locate all employee records for a given department and the department record in the same partition. This would allow department specific queries to process data on a single partition which can improve the overall system performance.

In example 1, the EmployeeKey class contains the deptId that is used to identify the department. By specifying the PartitionKey annotation on deptId, only the deptId field is used in the calculation of the partition. Similarly, when inserting department records using the DepartmentKey, only the deptId field is used in the calculation of the partition. This will allow employees and departments with the same deptId values to reside on the same partition.

Example 2 illustrates how it is also possible to reference fields in nested classes for performing the partitioning calculation. In this example, the EmployeeKey contains a reference to the DepartmentKey object. The PartitionKey annotation has a value equal to deptId which identifies the field in the DepartmentKey class that should be used by the partitioning calculation.

The PartitionKeys annotation is not illustrated here, but provides additional flexibility for defining which fields contribute to the partitioning calculation.

Class evolution and XDF

- Class evolution allows new fields to be added to a class without requiring all users of the class to change
- The data within the grid can evolve – allowing new applications to share existing data and enhance existing objects with new fields.
- XDF supports class evolution by storing a superset of the class data provided that developers follow some best practices :
  - Requires some planning from the application perspective since some fields may be uninitialized if the record was introduced with an older version of the application.
  - Applications need to use the update() API when altering existing records
  - Keys cannot be evolved using this technique
  - Do not change the type of a given field

Class evolution refers to the ability for a class to add new fields to support new applications or new versions of an application without requiring all applications to be updated at the same time.

Evolution may also include removing fields from a class definition, provided that the older versions of the application using this class are able to handle missing field data.

XDF uses the eXtreme Scale Mergeable interface for serializers to retain new fields during an update that are not provided by the caller of the update.

XDF supports class evolution by allowing users to keep a superset of the class data defined within the grid provided that the developers follow some best practices :

-Newer applications that add new fields from the common base class need to be able to handle uninitialized values after getting a value from the grid since the value may have been inserted by an older client that does not have the new fields.

-If an older version of an application has inserted a value into the grid and a newer application retrieves it, the value retrieved will be populated with a default value for the missing field as defined by the programming language (typically null or 0).

-Applications need to use the update method when altering existing records as opposed to delete and insert in order to ensure that new fields added by newer applications are not lost. Only the update method will retain fields that are not part of the local application's class definition.

-Keys cannot be evolved using this technique since the hashCode used for identifying a key will be based on the serialized bytes, which will be different if new fields are added to the key class.

-Do not change the type of a given field – changing a field definition from integer to string will result in casting errors when applications attempt to share the values

Class evolution and XDF – an Example (1 of 5)

- Consider two applications with different evolved versions of the classes.

SalesApp Customer defn

```
public class Customer {
   String name;
   Address addr;
   Date customerSince;
}
```

CMApp Customer defn

```
public class Customer {
   String name;
   Address addr;
   String rewardCardNumber;
}
```

Consider the following work flow:

(1) CMApp map.insert(1234, { Joe, Main St, rewardCard='Loyal111' }

(2) SalesApp map.insert(5678, {Bill, High St, customerSince='2000-03-04' })

Customer map

Customer id = 1234
name : Joe
Addr : Main St
rewardCard : Loyal111

Customer id = 5678
Name : Bill
Addr : High St
customerSince : 2000-03-04

Suppose that there are two applications with different evolved versions of the classes. The sales application named SalesApp existed first and a customer management application named CMApp was recently added.

Customer records are stored in the grid for the SalesApp and now you want to use the same map for the customer management application.

Notice that the customer management application introduces a new field for tracking the customer loyalty program number. From the reward card number, you can infer how long that person has been a customer and as a result, the customerSince field from the original version of the Customer class can be removed.

Regardless of which application inserts data into the grid, both applications will be able to read the Customer from the grid and work with the two fields that the applications have in common (name and address)

If the 'customer since' or 'reward card number' fields are null, then you need to retrieve that information from the permanent data store and then update the record.

Provided that all updates by the applications are performed using the Object Map update API, you will not lose any information provided by the other application for the additional field that the application does not know about.

Here is how it works.

The customer management application inserts a new record for customer id 1234 for Joe

The sales application inserts a new record for customer id 5678 for Bill.

There are now two customer objects in the grid with different definitions.

Class evolution and XDF – an Example (2 of 5)

- Consider two applications with different evolved versions of the classes.

SalesApp Customer defn

```
public class Customer {
   String name;
   Address addr;
   Date customerSince;
}
```

CMApp Customer defn

```
public class Customer {
   String name;
   Address addr;
   String rewardCardNumber;
}
```

Consider the following work flow:

(3) Joe moves – sales person updates his record with SalesApp
map.get(1234) – returns { Joe, Main St, customerSince=null }
map.update(1234, { Joe, Low St, customerSince='1999-03-04' })

Customer map

Customer id = 1234
name : Joe
Addr : Low St
rewardCard : Loyal111
customerSince : 1999-03-04

Customer id = 5678
Name : Bill
Addr : High St
customerSince : 2000-03-04

In Step 3, Joe moves from Main street to Low street and the sales person for his account updates his address with the SalesApp – since the customerSince field is not yet populated the application retrieves it from the database and updates the grid. Notice that you now have a logical Customer class that includes both the customerSince and rewardCardNumber fields even though there is no real class that contains both of these fields. Since the application performed an update, you did not lose the rewardCard information in the grid.

Class evolution and XDF – an Example (3 of 5)

- Consider two applications with different evolved versions of the classes.

SalesApp Customer defn

```
public class Customer {
   String name;
   Address addr;
   Date customerSince;
}
```

CMApp Customer defn

```
public class Customer {
   String name;
   Address addr;
   String rewardCardNumber;
}
```

Consider the following work flow:

Customer map

Customer id = 1234
name : Joe
Addr : Low St
rewardCard : Loyal111
customerSince : 1999-03-04

Customer id = 5678
Name : Bill
Addr : High St
customerSince : 2000-03-04

(4) CMApp retrieves Bill's record
map.get(5678) – returns { Bill, High St, rewardCard=null }

In step 4, the customer service team calls Bill and pulls up his record with the customer management application. The CMApp retrieves the record from the grid and sees that the rewardCard field is not populated.

Class evolution and XDF – an Example (4 of 5)

- Consider two applications with different evolved versions of the classes.

SalesApp Customer defn

```
public class Customer {
   String name;
   Address addr;
   Date customerSince;
}
```

CMApp Customer defn

```
public class Customer {
   String name;
   Address addr;
   String rewardCardNumber;
}
```

Consider the following work flow:

Customer map

Customer id = 1234
name : Joe
Addr : Low St
rewardCard : Loyal111
customerSince : 1999-03-04

(5) CMApp updates Bill's record
map.update(5678, {Bill, addr2, rewardCard='Loyal765' })

Customer id = 5678
Name : Bill
Addr : High St
customerSince : 2000-03-04
rewardCard : Loyal765

Step 5, the application then queries the database that stores the rewardCard information and updates the grid with the rewardCard information. Now Bill's record in the grid has both the rewardCard and customerSince values

Class evolution and XDF – an Example (5 of 5)

- Consider two applications with different evolved versions of the classes.

SalesApp Customer defn

```
public class Customer {
   String name;
   Address addr;
   Date customerSince;
}
```

CMApp Customer defn

```
public class Customer {
   String name;
   Address addr;
   String rewardCardNumber;
}
```

Consider the following work flow:

Customer map

Customer id = 1234
name : Joe
Addr : Low St
rewardCard : Loyal111
customerSince : 1999-03-04

Customer id = 5678
Name : Bill
Addr : Side St
customerSince : 2000-03-04

(6) Bill updates his address using the SalesWeb website
map.get(5678)
map.remove(5678)
map.insert(5678, {Bill, Side St, customerSince='2000-03-04' })

Step 6 - Consider what happens when an application does not follow the best practice of performing updates. Suppose that Bill moves and decides to use the SalesWeb customer website to update Bill's address. The SalesWeb site is coded to perform a remove and insert because the developer did not think about using the Object Map update API for this change. Notice that the value in the grid no longer contains the rewardCard information because the record in the grid was replaced.

C# and Java collection compatibility for generic collections (1 of 2)

- Generic in C# are enforced during runtime while Generics in Java are not
- Consider the following sample code from Java on left and C# on the right

```
class Customer {
    String name;
    ArrayList<Site> sites;
}

class Site {
    String siteName;
    String address;
    String city;
    String country;
    String postalCode;
}
```

```
class Customer
{
    string name;
    List<Site> sites;
}

class Site
{
    string siteName;
    string address;
    string city;
    string country;
    string postalCode;
}
```


The next two slides discuss interoperation considerations to be aware of when sharing collections between the C# and Java programming languages.

Generics in the C# programming language are enforced at runtime, while generics in the Java programming language are a compile time concept.

As a result there are different type identifiers assigned in the C# serialization of generic collections while Java collections of a particular type will all map to the same type identifier.

For example, a list of strings in C# will have a different type identifier than a list of customer objects. However in Java an ArrayList of strings and an ArrayList of customer objects will both be assigned the same type identifier internally.

Consider the two class definitions – the Java definition on the left and the C# definition on the right. Suppose there is a map of customers tracked by their customer ID. Within the customer record is a list of all the customer sites that have been delivered to, stored in a class called Site. There are both Java and C# applications that will perform operations on the Customer object and will access the list of site objects from C# and the ArrayList of site objects from Java.

When the Java application inserts a value, the ArrayList is serialized in such a way that it looks like a ArrayList of objects - the Java runtime does not enforce that only site objects can be placed into the ArrayList and so no attempt is made to validate all of the type information in XDF code since this can be expensive and impact performance.

When the C# application reads the customer record created by the Java application, the default conversion would be to put the site list into an ArrayList of objects in the C# language (note that ArrayLists are not generic in C#). However, since the C# runtime can provide the eXtreme Scale client with the target type that is expected, if the definition of the site list in the customer object is a list of site objects, then XDF can instantiate a list of site objects expected by the application.

C# and Java collection compatibility for generic collections (2 of 2)

- Suppose a grid just holds a list of sites (not embedded in the Customer object this time).
- Java stores as ArrayList<Site> and C# stores as List<Site>
- Code example 1 : Put list of Site into map using C# - map is defined for a List<Site> values

```
IGrid grid = getGrid("Grid");
List<Site> newSites = getNewSites();

IGridMapPessimisticAutoTx<Int32, List<Site>> custSiteMap =
    grid.GetGridMapPessimisticAutoTx<Int32, List<Site>>("CustSites");

custSiteMap.Add(10, newSites);
List<Site> sites = custSiteMap.Get(10);
```

- Code example 2 : Getting list of Site from map defined for Object values (not recommended)

```
IGrid grid = getGrid("Grid");
Object newSites = getNewSites();

IGridMapPessimisticAutoTx<Int32, Object> custSiteMap =
    grid.GetGridMapPessimisticAutoTx<Int32, Object>("CustSites");

List<Site> sites = (List<Site>)custSiteMap.Get(10);
```

14 eXtreme Data Format

Consider a slightly different example focusing on the C# code in particular – the Java processing is fairly straight forward – the type mappings from C# to Java are predefined and located in the documentation.

Suppose you are going to store a list of sites in the grid map. From the Java side, you use an ArrayList of site objects. On the C# side, you use a list of site objects.

Focus on the GetGridMapPessimisticAutoTx API call and in particular how generics are used to indicate that this map will have values of type List of Site objects.

In this situation, assume that the grid contains data that was inserted from a Java application as an ArrayList. When you retrieve the ArrayList value to C# with the custSiteMap get call, a conversion will occur from ArrayList to List because the map definition used generics to express the type that the application is expecting.

The second code example indicates that this map will contain values of type Object – so from the XDF point of view, you cannot use the map definition to convert the ArrayList to a List during the get call like you did in example 1.

As a result, if the same Java ArrayList was retrieved by C#, the XDF logic will instantiate a C# ArrayList since you do not know that the user is expecting a List of Sites.

The last line of code map in example 2 will fail in this situation with a cast exception since the ArrayList type cannot be cast to a list type. As a result, applications should be as specific as possible when working with lists to identify the target types on the getGridMap calls to allow XDF to perform the necessary conversions to the expected target type.

When is XDF Serialization not used ?

- Grid defines a DataSerializer plug-in
- XDF serialization may be partially disabled based on the class definitions in the following situations :
  - Java class implements Externalizable or the Serializable readObject or writeObject methods
  - C# class implements the ISerializable interface the getObjectData method
- In these situations all the nested classes must be serializable
- If XDF is not used, Java and C# cannot share the data serialized by the native serializer
- May be able to use class evolution to avoid this interoperation limitation

There are some situations where XDF serialization is partially or completely disabled.

-If a grid has defined data serializer plug-in then XDF is not used for serialization since you have indicated that you want to control the serialization. Note that data serializers are not supported from .NET, so maps defining a data serializer cannot be used by .NET applications.

-If a Java class implements the Externalizable interface or implements the Serializable interface's readObject and writeObject methods, then XDF is not used for that class and all of its nested children.

-If a C# class implements the ISerializable interface and the getObjectData method, then XDF is not used for that class and all of its nested children.

-In these situations, all of the nested classes must be serializable as defined by the programming languages interfaces.

-If Java and C# applications attempt to share data across the programming languages that are using these non-XDF serialization mechanisms, then an error will occur when attempting to de-serialize a value not serialized with XDF. One way this can be avoided if Java or C# serialization is necessary for one field is to use class evolution to only include that field in the class definition of the particular language where it is needed. For example, if you have a Java Externalizable class that contains the customer's purchasing preferences and this information is not needed for C#, then the C# application could exclude that field in the C# class definition. When the serialized value from Java is read by the C# application then the Java serialized field is silently ignored if there is no field with the same name to put the Java value.

Additional Information about XDF

- When a HashIndex is defined for a map it is more efficient to use fieldAccessAttribute="true" as this does not require the objects to be inflated on the server when indexing values. If fieldAccessAttribute="false" then the object will be inflated on the server as part of the insert or update logic that updates the index on the server.
- When indexing nested fields, XDF uses a "/" character to define the path to the fields to be accessed. For example, specify "addr/city" in the HashIndex definition in order to index the city from Address object that is embedded in the Employee object in the example below.

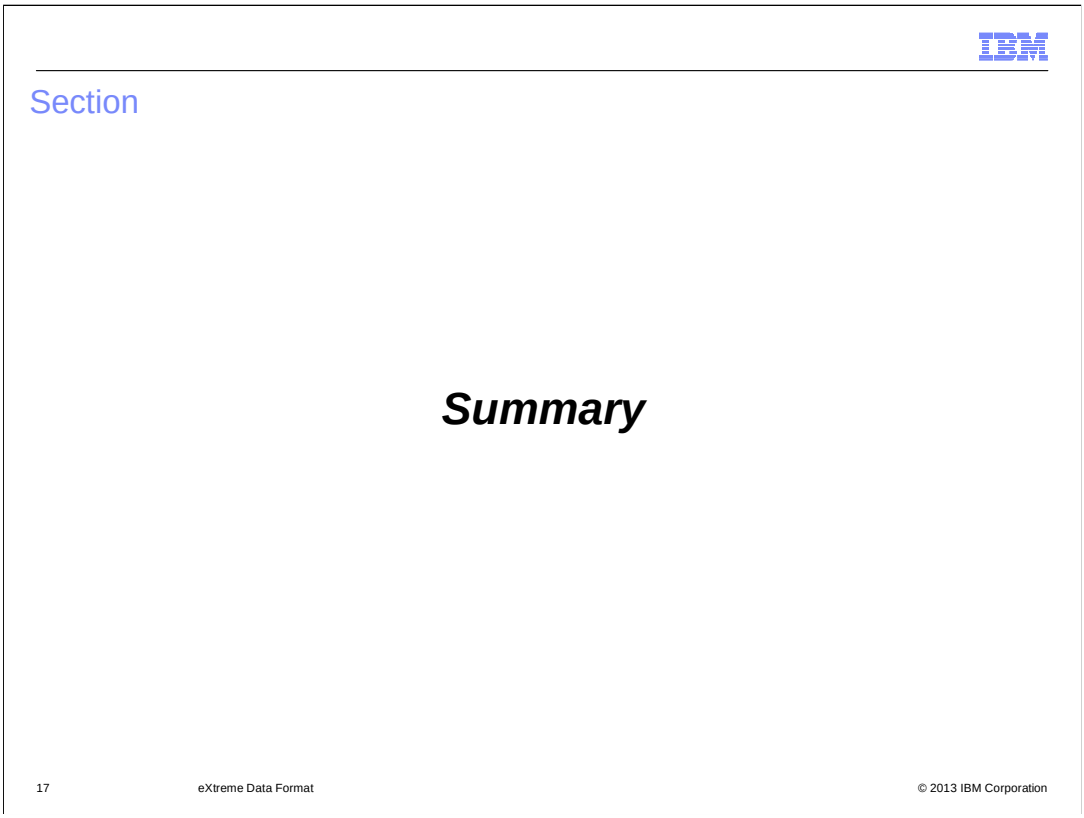
```
public class Employee {
    String name;
    Address addr;
}
```

```
public class Address {
    String street;
    String city;
    String country;
    String postalCode;
}
```

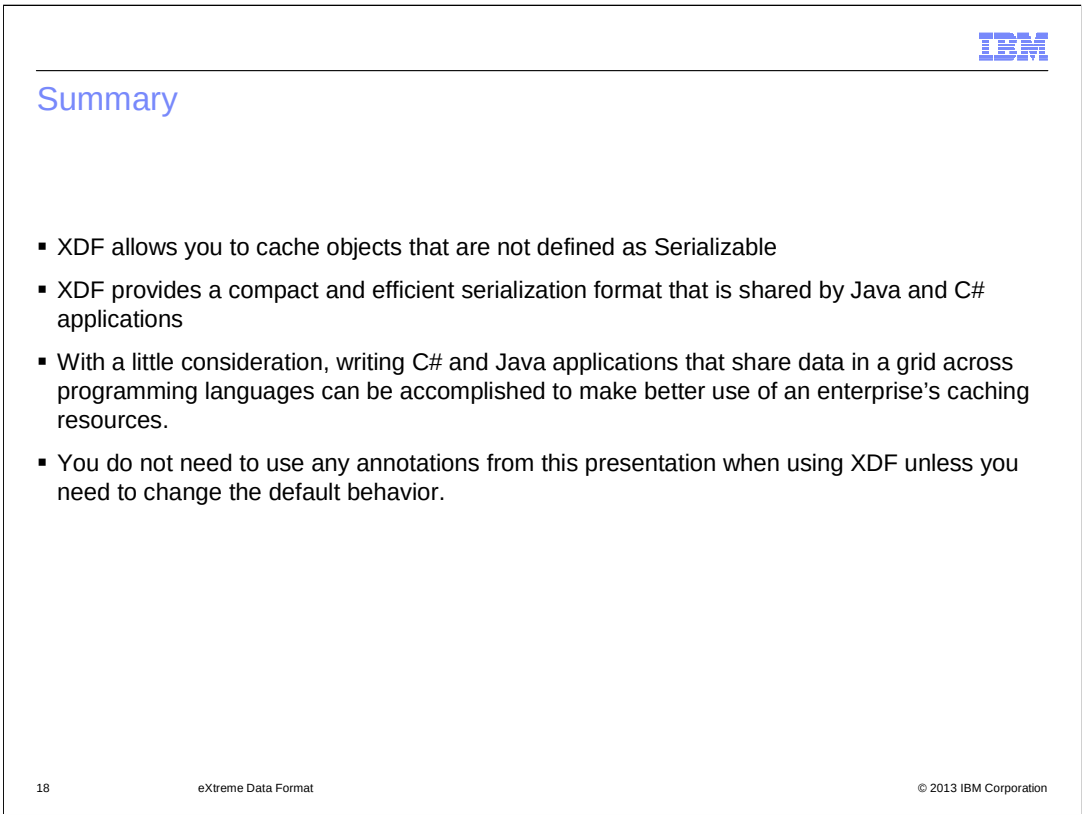
16 eXtreme Data Format

When a Hash Index is defined for a map, it is more efficient to use fieldAccessAttribute="true" because this does not require the objects to be inflated on the server when indexing values. If fieldAccessAttribute="false" then the object will be inflated on the server as part of the insert or update logic that updates the index on the server. This means that when working with C#, all Hash Indexes must be defined with fieldAccessAttribute="true" since the C# objects cannot be instantiated in the Java server runtime. This also means the user Java classes must be defined in the class path of all the container servers.

When indexing nested fields, the forward slash character is used to define the path to the fields that will be included in the index. For example, suppose you want to index the city field that exists within the Employee object in the sample class shown. You can reference the city field that is nested in the Address object by using the addr slash city path in the hash index definition.



The XDF feature provides many features that let you control partitioning, serialization, and interoperability between the Java and C# programming languages.



XDF provides features to serialize objects that are not defined as Serializable.

XDF provides a compact and efficient serialization format that is shared by Java and C# applications.

With a little consideration, writing C# and Java applications that share data in a grid across programming languages can be accomplished to make better use of an enterprise's caching resources.

You do not need to use any annotations from this presentation when using XDF, but they are there when you want to change the default behavior.

References

- WebSphere eXtreme Scale 8.6 Information Center
  http://pic.dhe.ibm.com/infocenter/wxsinfo/v8r6/index.jsp
- Configuring a grid to use XDF (Information Center)
  http://pic.dhe.ibm.com/infocenter/wxsinfo/v8r6/topic/com.ibm.websphere.extremescale.doc/txsconfigxdf.html

See these references for additional information about the eXtreme data format.

Feedback

Your feedback is valuable

You can help improve the quality of IBM Education Assistant content to better meet your needs by providing feedback.

1. Did you find this module useful?
2. Did it help you solve a problem or answer a question?
3. Do you have suggestions for improvements?

Click to send email feedback:

mailto:iea@us.ibm.com?subject=Feedback_about_XS86_XDF.ppt

This module is also available in PDF format at: ../XS86_XDF.pdf

You can help improve the quality of IBM Education Assistant content by providing feedback.

Trademarks, disclaimer, and copyright information