Customizing the Java Mapping

You can customize the code that the Slice-to-Java compiler produces by annotating your Slice definitions with metadata. This section describes how metadata influences several aspects of the generated Java code.

On this page:

Java Packages

By default, the scope of a Slice definition determines the package of its mapped Java construct. A Slice type defined in a module hierarchy is mapped to a type residing in the equivalent Java package.

There are times when applications require greater control over the packaging of generated Java classes. For instance, a company may have software development guidelines that require all Java classes to reside in a designated package. One way to satisfy this requirement is to modify the Slice module hierarchy so that the generated code uses the required package by default. In the example below, we have enclosed the original definition of `Workflow::Document` in the modules `com::acme` so that the compiler will create the class in the `com.acme` package:

Slice

```
module com {
    module acme {
        module Workflow {
            class Document {
                // ...
            };
        };
    };
};
```

There are two problems with this workaround:

1. It incorporates the requirements of an implementation language into the application's interface specification.
2. Developers using other languages, such as C++, are also affected.

The Slice-to-Java compiler provides a better way to control the packages of generated code through the use of global metadata. The example above can be converted as follows:

Slice

```
[["java:package:com.acme"]]
module Workflow {
    class Document {
        // ...
    };
};
```

The global metadata directive `java:package:com.acme` instructs the compiler to generate all of the classes resulting from definitions in this Slice file into the Java package `com.acme`. The net effect is the same: the class for `Document` is generated in the package `com.acme.Workflow`. However, we have addressed the two shortcomings of the first solution by reducing our impact on the interface specification: the Slice-to-Java compiler recognizes the package metadata directive and modifies its actions accordingly, whereas the compilers for other language mappings simply ignore it.

Java Package Configuration Properties

Using global metadata to alter the default package of generated classes has ramifications for the Ice run time when unmarshaling exceptions and concrete class types. The Ice run time dynamically loads generated classes by translating their Slice type ids into Java class names. For example, the Ice run time translates the Slice type id `::Workflow::Document` into the class name `Workflow.Document`.

However, when the generated classes are placed in a user-specified package, the Ice run time can no longer rely on the direct translation of a Slice type id into a Java class name, and therefore requires additional configuration so that it can successfully locate the generated classes. Two configuration properties are supported:

- `Ice.Package.`*`Module=package`*
  Associates a top-level Slice module with the package in which it was generated.

  > Only top-level module names are allowed; the semantics of global metadata prevent a nested module from being generated into a different package than its enclosing module.

- `Ice.Default.Package=package`
  Specifies a default package to use if other attempts to load a class have failed.

The behavior of the Ice run time when unmarshaling an exception or concrete class is described below:

1. Translate the Slice type id into a Java class name and attempt to load the class.
2. If that fails, extract the top-level module from the type id and check for an `Ice.Package` property with a matching module name. If found, prepend the specified package to the class name and try to load the class again.
3. If that fails, check for the presence of `Ice.Default.Package`. If found, prepend the specified package to the class name and try to load the class again.
4. If the class still cannot be loaded, the instance may be sliced.

Continuing our example from the previous section, we can define the following property:

```
Ice.Package.Workflow=com.acme
```

Alternatively, we could achieve the same result with this property:

```
Ice.Default.Package=com.acme
```

Custom Types in Java

One of the more powerful applications of metadata is the ability to tailor the Java mapping for sequence and dictionary types to match the needs of your application.

Metadata in Java

The metadata for specifying a custom type has the following format:

```
java:type:instance-type[:formal-type]
```

The formal type is optional; the compiler uses a default value if one is not defined. The instance type must satisfy an is-A relationship with the formal type: either the same class is specified for both types, or the instance type must be derived from the formal type.

The Slice-to-Java compiler generates code that uses the formal type for all occurrences of the modified Slice definition except when the generated code must instantiate the type, in which case the compiler uses the instance type instead.

The compiler performs no validation on your custom types. Misspellings and other errors will not be apparent until you compile the generated code.

Defining a Custom Sequence Type in Java

Although the default mapping of a sequence type to a native Java array is efficient and typesafe, it is not always the most convenient representation of your data. To use a different representation, specify the type information in a metadata directive, as shown in the following example:

Slice

```
["java:type:java.util.LinkedList<String>"]
sequence<string> StringList;
```

It is your responsibility to use a type parameter for the Java class (`String` in the example above) that is the correct mapping for the sequence's element type.

The compiler requires the formal type to implement `java.util.List<`*E*`>`, where *E* is the Java mapping of the element type. If you do not specify a formal type, the compiler uses `java.util.List<`*E*`>` by default.

Note that extra care must be taken when defining custom types that contain nested generic types, such as a custom sequence whose element type is also a custom sequence. The Java compiler strictly enforces type safety, therefore any compatibility issues in the custom type metadata will be apparent when the generated code is compiled.

Defining a Custom Dictionary Type in Java

The default instance type for a dictionary is `java.util.HashMap<`*K*`, `*V*`>`, where *K* is the Java mapping of the key type and *V* is the Java mapping of the value type. If the semantics of a `HashMap` are not suitable for your application, you can specify an alternate type using metadata as shown in the example below:

Slice

```
["java:type:java.util.TreeMap<String, String>"]
dictionary<string, string> StringMap;
```

It is your responsibility to use type parameters for the Java class (`String` in the example above) that are the correct mappings for the dictionary's key and value types.

The compiler requires the formal type to implement `java.util.Map<`*K*`, `*V*`>`. If you do not specify a formal type, the compiler uses this type by default.

Note that extra care must be taken when defining dictionary types that contain nested generic types, such as a dictionary whose element type is a custom sequence. The Java compiler strictly enforces type safety, therefore any compatibility issues in the custom type metadata will be apparent when the generated code is compiled.

Using Custom Type Metadata in Java

You can define custom type metadata in a variety of situations. The simplest scenario is specifying the metadata at the point of definition:

Slice

```
["java:type:java.util.LinkedList<String>"]
sequence<string> StringList;
```

Defined in this manner, the Slice-to-Java compiler uses `java.util.List<String>` (the default formal type) for all occurrences of `StringList`, and `java.util.LinkedList<String>` when it needs to instantiate `StringList`.

You may also specify a custom type more selectively by defining metadata for a data member, parameter or return value. For instance, the mapping for the original Slice definition might be sufficient in most situations, but a different mapping is more convenient in particular cases. The example below demonstrates how to override the sequence mapping for the data member of a structure as well as for several operations:

Slice

```
sequence<string> StringSeq;

struct S {
    ["java:type:java.util.LinkedList<String>"] StringSeq seq;
};

interface I {
    ["java:type:java.util.ArrayList<String>"] StringSeq
    modifiedReturnValue();

    void modifiedInParam(["java:type:java.util.ArrayList<String>"] StringSeq seq);

    void modifiedOutParam(out ["java:type:java.util.ArrayList<String>"] StringSeq seq);
};
```

As you might expect, modifying the mapping for an operation's parameters or return value may require the application to manually convert values from the original mapping to the modified mapping. For example, suppose we want to invoke the `modifiedInParam` operation. The signature of its proxy operation is shown below:

Java

```
void modifiedInParam(java.util.List<String> seq, Ice.Current curr)
```

The metadata changes the mapping of the `seq` parameter to `java.util.List`, which is the default formal type. If a caller has a `StringSeq` value in the original mapping, it must convert the array as shown in the following example:

Java

```
String[] seq = new String[2];
seq[0] = "hi";
seq[1] = "there";
IPrx proxy = ...;
proxy.modifiedInParam(java.util.Arrays.asList(seq));
```

Although we specified the instance type `java.util.ArrayList<String>` for the parameter, we are still able to pass the result of `asList` because its return type (`java.util.List<String>`) is compatible with the parameter's formal type declared by the proxy method. In the case of an operation parameter, the instance type is only relevant to a servant implementation, which may need to make assumptions about the actual type of the parameter.

Mapping for Modified Out Parameters in Java

The mapping for an `out` parameter uses a generated "holder" class to convey the parameter value. If you modify the mapping of an `out` parameter, as discussed in the previous section, it is possible that the holder class for the parameter's unmodified type is no longer compatible with the custom type you have specified. The holder class generated for `StringSeq` is shown below:

Java

```
public final class StringSeqHolder
{
    public
    StringSeqHolder()
    {
    }

    public
    StringSeqHolder(String[] value)
    {
        this.value = value;
    }

    public String[] value;
}
```

An `out` parameter of type `StringSeq` would normally map to a proxy method that used `StringSeqHolder` to hold the parameter value. When the parameter is modified, as is the case with the `modifiedOutParam` operation, the Slice-to-Java compiler cannot use `StringSeqHolder` to hold an instance of `java.util.List<String>`, because `StringSeqHolder` is only appropriate for the default mapping to a native array.

As a result, the compiler handles these situations using instances of the generic class `Ice.Holder<`*T*`>`, where *T* is the parameter's formal type. Consider the following example:

Slice

```
sequence<string> StringSeq;

interface I {
    void modifiedOutParam(out ["java:type:java.util.ArrayList<String>"] StringSeq seq);
};
```

The compiler generates the following mapping for the `modifiedOutParam` proxy method:

Java

```
void modifiedOutParam(Ice.Holder<java.util.List<java.lang.String> > seq, Ice.Current curr)
```

The formal type of the parameter is `java.util.List<String>`, therefore the holder class becomes `Ice.Holder<java.util.List<String>>`.

JavaBean Mapping

The Java mapping optionally generates JavaBean-style methods for the data members of class, structure, and exception types.

JavaBean Generated Methods

For each data member *val* of type *T*, the mapping generates the following methods:

Java

```
public T getVal();
public void setVal(T v);
```

The mapping generates an additional method if *T* is the `bool` type:

Java

```
public boolean isVal();
```

Finally, if *T* is a sequence type with an element type *E*, two methods are generated to provide direct access to elements:

Java

```
public E getVal(int index);
public void setVal(int index, E v);
```

Note that these element methods are only generated for sequence types that use the default mapping.

The Slice-to-Java compiler considers it a fatal error for a JavaBean method of a class data member to conflict with a declared operation of the class. In this situation, you must rename the operation or the data member, or disable the generation of JavaBean methods for the data member in question.

JavaBean Metadata

The JavaBean methods are generated for a data member when the member or its enclosing type is annotated with the `java:getset` metadata. The following example demonstrates both styles of usage:

Slice

```
sequence<int> IntSeq;

class C {
    ["java:getset"] int i;
    double d;
};

["java:getset"]
struct S {
    bool b;
    string str;
};

["java:getset"]
exception E {
    IntSeq seq;
};
```

JavaBean methods are generated for all members of struct `S` and exception `E`, but for only one member of class `C`. Relevant portions of the generated code are shown below:

Java

```
public class C extends Ice.ObjectImpl
{
    ...

    public int i;

    public int
    getI()
    {
        return i;
    }

    public void
```

```
    setI(int _i)
    {
        i = _i;
    }

    public double d;
}

public final class S implements java.lang.Cloneable
{
    public boolean b;

    public boolean
    getB()
    {
        return b;
    }

    public void
    setB(boolean _b)
    {
        b = _b;
    }

    public boolean
    isB()
    {
        return b;
    }

    public String str;

    public String
    getStr()
    {
        return str;
    }

    public void
    setStr(String _str)
    {
        str = _str;
    }

    ...
}

public class E extends Ice.UserException
{
    ...

    public int[] seq;

    public int[]
    getSeq()
    {
        return seq;
    }

    public void
    setSeq(int[] _seq)
    {
        seq = _seq;
    }

    public int
    getSeq(int _index)
```

```
    {
        return seq[_index];
    }

    public void
    setSeq(int _index, int _val)
    {
        seq[_index] = _val;
    }

    ...
}
```

Overriding `serialVersionUID`

The Slice-to-Java compiler computes a default value for the `serialVersionUID` member of Slice classes, exceptions and structures. If you prefer, you can override this value using the `java:serialVersionUID` metadata, as shown below:

Slice

```
["java:serialVersionUID:571254925"]
struct Identity
{
    ...
};
```

The specified value will be used in place of the default value in the generated code:

Java

```
public class Identity
{
    ...

    public static final long serialVersionUID = 571254925L;
}
```

By using this metadata, the application assumes responsibility for updating the UID whenever changes to the Slice definition affect the serializable state of the type.

See Also

- Metadata
- Java Mapping for Modules
- Java Mapping for Operations
- Class Inheritance Semantics