Upgrading your Application from Ice 3.3

In addition to the information provided in Upgrading your Application from Ice 3.4, users who are upgrading from Ice 3.3 should also review this page.

On this page:

Backward compatibility of Ice versions

A discussion of backward compatibility in Ice involves many factors.

Source-code compatibility

Ice maintains source-code compatibility between a patch release (e.g., 3.4.2) and the most recent minor release (e.g., 3.4.0), but does not guarantee source-code compatibility between minor releases (e.g., between 3.4 and 3.5).

The subsections below describe the significant API changes in this release that may impact source-code compatibility. Furthermore, the subsections Removed APIs in Ice 3.4.0 and Deprecated APIs in Ice 3.4.0 summarize additional changes to Ice APIs that could affect your application.

Binary compatibility

As for source-code compatibility, Ice maintains backward binary compatibility between a patch release and the most recent minor release, but does not guarantee binary compatibility between minor releases.

The requirements for upgrading depend on the language mapping used by your application:

- For statically-typed languages (C++, Java, .NET), the application must be recompiled.
- For scripting languages that use static translation, your Slice files must be recompiled.
- No action is necessary for a Python or Ruby script that loads its Slice files dynamically.

On-the-wire compatibility

Ice always maintains protocol ("on the wire") compatibility with prior releases. A client using Ice version *x* can communicate with a server using Ice version *y* and vice versa.

Several features introduced in Ice 3.5 require a new version of the Ice encoding, encoding version 1.1. Older versions of Ice do not understand this encoding: you need to use Ice encoding version 1.0 for communications between clients or servers using Ice 3.5 and clients and servers using older Ice versions. See Encoding Version 1.1 for details.

Database compatibility

Upgrading to a new minor release of Ice often includes an upgrade to the supported version of Berkeley DB. In turn, this may require an application to migrate its databases, either because the format of Berkeley DB's database files has changed, or due to a change in the schema of the data stored in those databases.

For example, if your application uses Freeze, it may be necessary for you to migrate your databases even if your schema has not changed.

Certain Ice services also use Freeze in their implementation. If your application uses these services (IceGrid and IceStorm), it may be necessary for you to migrate their databases as well.

Please refer to the relevant subsections below for migration instructions.

Interface compatibility

Although Ice always maintains compatibility at the protocol level, changing Slice definitions can also lead to incompatibilities. As a result, Ice maintains interface compatibility between a patch release and the most recent minor release, but does not guarantee compatibility between minor releases.

This issue is particularly relevant if your application uses Ice services such as IceGrid or IceStorm, as a change to an interface in one of these services may adversely affect your application.

Interface changes in an Ice service can also impact compatibility with its administrative tools, which means it may not be possible to administer an Ice 3.4.x service using a tool from a previous minor release (or vice-versa).

IceGrid

Starting with Ice 3.2.0, IceGrid registries and nodes are interface-compatible. For example, you can use an IceGrid node from Ice 3.2 with a registry from Ice 3.4.

IceGrid registry replication is only supported between registries using Ice 3.3 or later.

An IceGrid node using Ice 3.3 or later is able to activate a server that uses Ice 3.2. The reverse is also true: an IceGrid node from Ice 3.2 is able to activate a server built with Ice 3.3 or later, but only if the server's configuration properties do not rely on features added after Ice 3.2 (such as the ability to escape characters in property names and values).

IceStorm

Topic linking is supported between all IceStorm versions released after 3.0.0.

Java language mapping changes in Ice 3.4

The Java2 language mapping, which was deprecated in Ice 3.3, is no longer supported. The Slice compiler and Ice API now use the Java5 language mapping exclusively, therefore upgrading to Ice 3.4 may require modifications to your application's source code. The subsections below discuss the language mapping features that are affected by this change and describe how to modify your application accordingly.

Metadata

The global metadata directives `java:java2` and `java:java5` are no longer supported and should be removed from your Slice files. The Slice compiler now emits a warning about these directives.

Support for the portable metadata syntax has also been removed. This syntax allowed Slice definitions to define custom type metadata that the Slice compiler would translate to match the desired target mapping. For example:

Slice

```
["java:type:{java.util.ArrayList}"] sequence<String> StringList;
```

The braces surrounding the custom type `java.util.ArrayList` directed the Slice compiler to use `java.util.ArrayList<String>` in the Java5 mapping and `java.util.ArrayList` in the Java2 mapping.

All uses of the portable metadata syntax must be changed to use the corresponding Java5 equivalent.

Dictionaries

Now that Slice dictionary types use the Java5 mapping, recompiling your Slice files and your application may cause the Java compiler to emit "unchecked" warnings. This occurs when your code attempts to assign an untyped collection class such as `java.util.Map` to a generic type such as `java.util.Map<String, String>`. Consider the following example:

Slice

```
dictionary<string, int> ValueMap;

interface Table
{
    void setValues(ValueMap m);
};
```

A Java2 application might have used these Slice definitions as shown below:

Java

```
java.util.Map values = new java.util.HashMap();
values.put(...);

TablePrx proxy = ...;
proxy.setValues(values); // Warning
```

The call to `setValues` is an example of an unchecked conversion. We recommend that you compile your application using the compiler option shown below:

```
javac -Xlint:unchecked ...
```

This option causes the compiler to generate descriptive warnings about occurrences of unchecked conversions to help you find and correct the offending code.

Request Contexts

The Slice type for request contexts, `Ice::Context`, is defined as follows:

Slice

```
module Ice
{
    dictionary<string, string> Context;
};
```

As a dictionary, the `Context` type is subject to the same issues regarding unchecked conversions described for #Dictionaries. For example, each proxy operation maps to two overloaded methods, one that omits the trailing `Context` parameter and one that includes it:

Java

```
interface TablePrx
{
    void setValues(java.util.Map<String, Integer> m);  // No context

    void setValues(java.util.Map<String, Integer> m,
                   java.util.Map<String, String> ctx);
}
```

If your proxy invocations make use of this parameter, you will need to change your code to use the generic type shown above in order to eliminate unchecked conversion warnings.

Enumerations

The Java2 language mapping for a Slice enumeration generated a class whose API differed in several ways from the standard Java5 enum type. Consider the following enumeration:

Slice

```
enum Color { red, green, blue };
```

The Java2 language mapping for `Color` is shown below:

Java

```
public final class Color
{
    // Integer constants
    public static final int _red = 0;
    public static final int _green = 1;
    public static final int _blue = 2;

    // Enumerators
    public static final Color red = ...;
    public static final Color green = ...;
    public static final Color blue = ...;

    // Helpers
    public static Color convert(int val);
    public static Color convert(String val);
    public int value();

    ...
}
```

The first step in migrating to the Java5 mapping for enumerations is to modify all `switch` statements that use an enumerator. Before Java added native support for enumerations, the `switch` statement could only use the integer value of the enumerator and therefore the Java2 mapping supplied integer constants for use in `case` statements. For example, here is a `switch` statement that uses the Java2 mapping:

Java

```
Color col = ...;
switch(col.value())
{
case Color._red:
    ...
    break;
case Color._green:
    ...
    break;
case Color._blue:
    ...
    break;
}
```

The Java5 mapping eliminates the integer constants because Java5 allows enumerators to be used in `case` statements. The resulting code becomes much easier to read and write:

Java

```
Color col = ...;
switch(col)
{
case red:
    ...
    break;
case green:
    ...
    break;
case blue:
    ...
    break;
}
```

The next step is to replace any uses of the `value` or `convert` methods with their Java5 equivalents. The base class for all Java5 enumerations (`java.lang.Enum`) supplies methods with similar functionality:

Java

```
static Color[] values()          // replaces convert(int)
static Color valueOf(String val) // replaces convert(String)
int ordinal()                    // replaces value()
```

For example, here is the Java5 code to convert an integer into its equivalent enumerator:

Java

```
Color r = Color.values()[0]; // red
```

Note however that the `convert(String)` method in the Java2 mapping returned null for an invalid argument, whereas the Java5 enum method `valueOf(String)` raises `IllegalArgumentException` instead.

Refer to the manual for more details on the mapping for enumerations.

Changes to the Java API for Freeze maps in Ice 3.4

The Java API for Freeze maps has been revised to used Java5 generic types and enhanced to provide additional functionality. This section describes these changes in detail and explains how to migrate your Freeze application to the API in Ice 3.4.

General changes to Freeze maps in Java

The Freeze API is now entirely type-safe, which means compiling your application against Ice 3.4 is likely to generate unchecked conversion warnings. The generated class for a Freeze map now implements the `java.util.SortedMap<K, V>` interface, where *K* is the key type and *V* is the value type. As a result, applications that relied on the untyped `SortedMap` API (where all keys and values were treated as instances of `java.lang.Object`) will encounter compiler warnings in Ice 3.4.

For example, an application might have iterated over the entries in a map as follows:

Java

```
// Old API
Object key = new Integer(5);
Object value = new Address(...);
myMap.put(key, value);
java.util.Iterator i = myMap.entrySet().iterator();
while (i.hasNext())
{
    java.util.Map.Entry e = (java.util.Map.Entry)i.next();
    Integer myKey = (Integer)e.getKey();
    Address myValue = (Address)e.getValue();
    ...
}
```

This code will continue to work, but the new API is both type-safe and self-documenting:

Java

```
// New API
int key = 5;
Address value = new Address(...);
myMap.put(key, value); // The key is autoboxed to Integer.
for (java.util.Map.Entry<Integer, Address> e : myMap.entrySet())
{
    Integer myKey = e.getKey();
    Address myValue = e.getValue();
    ...
}
```

Although migrating to the new API may require some effort, the benefits are worthwhile because your code will be easier to read and less prone to defects. You can also take advantage of the "autoboxing" features in Java5 that automatically convert values of primitive types (such as `int`) into their object equivalents (such as `Integer`).

Please refer to the manual for complete details on the new API.

Enhancements to Freeze maps in Java

Java6 introduced the `java.util.NavigableMap` interface, which extends `java.util.SortedMap` to add some useful new methods. Although the Freeze map API cannot implement `java.util.NavigableMap` directly because Freeze must remain compatible with Java5, we have added the `Freeze.NavigableMap` interface to provide much of the same functionality. A generated Freeze map class implements `NavigableMap`, as do the sub map views returned by map methods such as `headMap`. The `NavigableMap` interface is described in the manual, and you can also refer to the Java6 API documentation.

Backward compatibility for Freeze maps in Java

The Freeze Map API related to indices underwent some significant changes in order to improve type safety and avoid unchecked conversion warnings. These changes may cause compilation failures in a Freeze application.

In the previous API, index comparator objects were supplied to the Freeze map constructor in a map (in Java5 syntax, this comparators map would have the type `java.util.Map<String, java.util.Comparator>`) in which the index name was the key. As part of our efforts to improve type safety, we also wanted to use the fully-specified type for each index comparator (such as `java.util.Comparator<Integer>`). However, given that each index could potentially use a different key type, it is not possible to retain the previous API while remaining type-safe.

Consequently, the index comparators are now supplied as data members of a static nested class of the Freeze map named `IndexComparators`. If your application supplied custom comparators for indices, you will need to revise your code to use `IndexComparators` instead. For example:

Java

```
// Old API
java.util.Map indexComparators = new java.util.HashMap();
indexComparators.put("index", new MyComparator());
MyMap map = new MyMap(..., indexComparators);

// New API
MyMap.IndexComparators indexComparators = new MyMap.IndexComparators();
indexComparators.valueComparator = new MyComparator();
MyMap map = new MyMap(..., indexComparators);
```

We also encourage you to modify the definition of your comparator classes to use the Java5 syntax, as shown in the example below:

Java

```
// Old comparator
class IntComparator implements java.util.Comparator
{
    public int compare(Object o1, Object o2)
    {
        return ((Integer)o1).compareTo(o2);
    }
}

// New comparator
class IntComparator implements java.util.Comparator<Integer>
{
    public int compare(Integer i1, Integer i2)
    {
        return i1.compareTo(i2);
    }
}
```

The second API change that might cause compilation failures is the removal of the following methods:

Java

```
java.util.SortedMap headMapForIndex(String name, Object key);
java.util.SortedMap tailMapForIndex(String name, Object key);
java.util.SortedMap subMapForIndex(String name, Object from, Object to);
java.util.SortedMap mapForIndex(String name);
```

Again, this API cannot be retained in a type-safe fashion, therefore `slice2freezej` now generates equivalent (and type-safe) methods for each index in the Freeze map class.

Please refer to the manual for complete details on the new API.

Finalizers in Freeze

In previous releases, Freeze for Java used finalizers to close objects such as maps and connections that the application neglected to close. Most of these finalizers have been removed in Ice 3.4, and the only remaining finalizers simply log warning messages to alert you to the fact that connections and iterators are not being closed explicitly. Note that, given the uncertain nature of Java finalizers, it is quite likely that the remaining finalizers will not be executed.

Freeze packaging changes in Ice 3.4

All Freeze-related classes are now stored in a separate JAR file named `Freeze.jar`. As a result, you may need to update your build scripts, deployment configuration, and run-time environment to include this additional JAR file.

PHP changes in Ice 3.4

The Ice extension for PHP has undergone many changes in this release. The subsections below describe these changes in detail. Refer to the PHP Mapping for more information about the language mapping.

Static translation in PHP

In prior releases, Slice files were deployed with the application and loaded at Web server startup by the Ice extension. Before each page request, the extension directed the PHP interpreter to parse the code that was generated from the Slice definitions.

In this release, Slice files must be translated using the new compiler `slice2php`. This change offers several advantages:

- Applications may have more opportunities to improve performance through the use of opcode caching.
- It is no longer necessary to restart the Web server when you make changes to your Slice definitions, which is especially useful during development.
- Errors in your Slice files can now be discovered in your development environment, rather than waiting until the Web server reports a failure and then reviewing the server log to determine the problem.
- The development process becomes simpler because you can easily examine the generated code if you have questions about the API or language mapping rules.
- PHP scripts can now use all of the Ice local exceptions. In prior releases, only a subset of the local exception types were available, and all others were mapped to `Ice_UnknownLocalException`. See the section Run-time exceptions in PHP below for more information.

All of the Slice files for Ice and Ice services are translated during an Ice build and available for inclusion in your application. At a minimum, you must include the file `Ice.php`:

PHP

```
require 'Ice.php';
```

`Ice.php` contains definitions for core Ice types and includes a minimal set of generated files. To use an Ice service such as IceStorm, include the appropriate generated file:

PHP

```
require 'Ice.php';
require 'IceStorm/IceStorm.php';
```

Deploying a PHP application

With the transition to static code generation, you no longer need to deploy Slice files with your application. Instead, you will need to deploy the PHP code generated from your Slice definitions, along with `Ice.php`, the generated code for the Ice core, and the generated code for any Ice services your application might use.

Using communicators in PHP

In prior releases, each PHP page request could access a single Ice communicator via the `$ICE` global variable. The configuration of this communicator was derived from the profile that the script loaded via the `Ice_loadProfile` function. The communicator was created on demand when `$ICE` was first used and destroyed automatically at the end of the page request.

In this release, a PHP script must create its own communicator using an API that is similar to other Ice language mappings:

PHP

```
function Ice_initialize()
function Ice_initialize($args)
function Ice_initialize($initData)
function Ice_initialize($args, $initData)
```

`Ice_initialize` creates a new communicator using the configuration provided in the optional arguments. `$args` is an array of strings representing command-line options, and `$initData` is an instance of `Ice_InitializationData`.

An application that requires no configuration can initialize a communicator as follows:

PHP

```
$communicator = Ice_initialize();
```

More elaborate configuration scenarios are described in the section #PHP configuration below.

A script may optionally destroy its communicator:

PHP

```
$communicator->destroy();
```

At the completion of a page request, Ice by default automatically destroys any communicator that was not explicitly destroyed.

Using registered communicators in PHP

PHP applications may benefit from the ability to use a communicator instance in multiple page requests. Reusing a communicator allows the application to minimize the overhead associated with the communicator lifecycle, including such activities as opening and closing connections to Ice servers.

This release includes new APIs for registering a communicator in order to prevent Ice from destroying it automatically at the completion of a page request. For example, a session-based application can create a communicator, establish a Glacier2 session, and register the communicator. In subsequent page requests, the PHP session can retrieve its communicator instance and continue using the Glacier2 session.

The manual provides more information on this feature, and a new sample program can be found in `Glacier2/hello`.

PHP configuration

Prior releases supported four INI settings in PHP's configuration file:

- `ice.config`
- `ice.options`
- `ice.profiles`
- `ice.slice`

The `ice.slice` directive is no longer supported since Slice definitions are now compiled statically. The remaining options are still supported but their semantics are slightly different. They no longer represent the configuration of a communicator; instead, they define property sets that a script can retrieve and use to initialize a communicator.

The global INI directives `ice.config` and `ice.options` configure the default property set. The `ice.profiles` directive can optionally nominate a separate file that defines any number of named profiles, each of which configures a property set.

As before, the profiles use an INI file syntax:

```
[Name1]
config=file1
options="--Ice.Trace.Network=2 ..."

[Name2]
config=file2
options="--Ice.Trace.Locator=1 ..."
```

A new directive, `ice.hide_profiles`, overwrites the value of the `ice.profiles` directive as a security measure. This directive has a default value of 1, meaning it is enabled by default.

A script can obtain a property set using the new function `Ice_getProperties`. Called without an argument (or with an empty string), the function returns the default property set:

PHP

```
$props = Ice_getProperties();
```

Alternatively, you can pass the name of the desired profile:

PHP

```
$props = Ice_getProperties("Name1");
```

The returned object is an instance of `Ice_Properties`, which supports the standard Ice API.

For users migrating from an earlier release, you can replace a call to `Ice_loadProfile` as follows:

PHP

```
// PHP - Old API
Ice_loadProfile('Name1');

// PHP - New API
$initData = new Ice_InitializationData;
$initData->properties = Ice_getProperties('Name1');
$ICE = Ice_initialize($initData);
```

(Note that it is not necessary to use the symbol `$ICE` for your communicator. However, using this symbol may ease your migration to this release.)

`Ice_loadProfile` also installed the PHP definitions corresponding to your Slice types. In this release you will need to add `require` statements to include your generated code.

Finally, if you wish to manually configure a communicator, you can create a property set using `Ice_createProperties`:

PHP

```
function Ice_createProperties($args=null, $defaultProperties=null)
```

`$args` is an array of strings representing command-line options, and `$defaultProperties` is an instance of `Ice_Properties` that supplies default values for properties.

As an example, an application can configure a communicator as shown below:

PHP

```
$initData = new Ice_InitializationData;
$initData->properties = Ice_createProperties();
$initData->properties->setProperty("Ice.Trace.Network", "1");
...
$ICE = Ice_initialize($initData);
```

PHP namespaces

This release includes optional support for PHP namespaces, which was introduced in PHP 5.3. Support for PHP namespaces is disabled by default; to enable it, you must build the Ice extension from source code with `USE_NAMESPACES=yes` (see `Make.rules` or `Make.rules.mak` in the `php/config` subdirectory). Note that the extension only supports one mapping style at a time; installing a namespace-enabled version of the extension requires all Ice applications on the target Web server to use namespaces.

With namespace support enabled, you must modify your script to include a different version of the core Ice types:

PHP

```
require 'Ice_ns.php'; // Namespace version of Ice.php
```

You must also recompile your Slice files using the `-n` option to generate namespace-compatible code:

```
% slice2php -n MySliceFile.ice
```

This mapping translates Slice modules into PHP namespaces instead of using the "flattened" (underscore) naming scheme. For example, `Ice_Properties` becomes `\Ice\Properties` in the namespace mapping. However, applications can still refer to global Ice functions by their traditional names (such as `Ice_initialize`) or by their namespace equivalents (`\Ice\initialize`).

Run-time exceptions in PHP

As mentioned earlier, prior releases of Ice for PHP only supported a limited subset of the standard run-time exceptions. An occurrence of an unsupported local exception was mapped to `Ice_UnknownLocalException`.

This release adds support for all local exceptions, which allows an application to more easily react to certain types of errors:

PHP

```
try
{
    $proxy->sayHello();
}
catch(Ice_ConnectionLostException $ex)
{
    // Handle connection loss
}
catch(Ice_LocalException $ex)
{
    // Handle other errors
}
```

This change represents a potential backward compatibility issue: applications that previously caught `Ice_UnknownLocalException` may need to be modified to catch the intended exception instead.

Downcasting in PHP

In prior releases, to downcast a proxy you had to invoke the `ice_checkedCast` or `ice_uncheckedCast` method on a proxy and supply a type ID:

PHP

```
$hello = $proxy->ice_checkedCast("::Demo::Hello");
```

This API is susceptible to run-time errors because no validation is performed on the type ID string. For example, renaming the `Hello` interface to `Greeting` requires that you not only change all occurrences of `Demo_Hello` to `Demo_Greeting`, but also fix any type ID strings that your code might have embedded. The PHP interpreter does not provide any assistance if you forget to make this change, and you will only discover it when that particular line of code is executed and fails.

To improve this situation, a minimal class is now generated for each proxy type. The purpose of this class is to supply `checkedCast` and `uncheckedCast` static methods:

PHP

```
class Demo_HelloPrx
{
    public static function checkedCast($proxy, $facetOrCtx=null, $ctx=null);

    public static function uncheckedCast($proxy, $facet=null);
}
```

Now your application can downcast a proxy as follows:

PHP

```
$hello = Demo_HelloPrx::checkedCast($proxy);
```

You can continue to use `ice_checkedCast` and `ice_uncheckedCast` but we recommend migrating your application to the new methods.

Other API changes for PHP

This section describes additional changes to the Ice API in this release:

- The global variable `$ICE` is no longer defined. An application must now initialize its own communicator as described above.
- Removed the following communicator methods:

  PHP

  ```
  $ICE->setProperty()
  $ICE->getProperty()
  ```

  The equivalent methods are:

  PHP

  ```
  $communicator->getProperties()->setProperty()
  $communicator->getProperties()->getProperty()
  ```

- Removed the following global functions:

  PHP

  ```
  Ice_stringToIdentity()
  Ice_identityToString()
  ```

The equivalent methods are:

PHP

```
$communicator->stringToIdentity()
$communicator->identityToString()
```

- These functions have also been removed:

PHP

```
Ice_loadProfile()
Ice_loadProfileWithArgs()
Ice_dumpProfile()
```

Refer to PHP configuration for more information.

Thread pool changes in Ice 3.4

A thread pool supports the ability to automatically grow and shrink as the demand for threads changes, within the limits set by the thread pool's configuration. In prior releases, the rate at which a thread pool shrinks was not configurable, but Ice 3.4.0 introduces the `ThreadIdleTime` property to allow you to specify how long a thread pool thread must remain idle before it terminates to conserve resources.

IceSSL changes in Ice 3.4

With the addition of the `ConnectionInfo` classes in this release, the `IceSSL::ConnectionInfo` structure has changed from a native type to a Slice class. This change has several implications for existing applications:

- As a Slice class, `IceSSL::ConnectionInfo` cannot provide the X509 certificate chain in its native form, therefore the chain is provided as a sequence of strings representing the encoded form of each certificate. You can use language-specific facilities to convert these strings back to certificate objects.
- For your convenience, we have added a native subclass of `IceSSL::ConnectionInfo` called `IceSSL::NativeConnectionInfo`. This class provides the certificate chain as certificate objects.
- The `CertificateVerifier` interface now uses `NativeConnectionInfo` instead of `ConnectionInfo`. If your application configures a custom certificate verifier, you will need to modify your implementation accordingly.
- In C++, also note that `NativeConnectionInfo`is now managed by a smart pointer, therefore the signature of the certificate verifier method becomes the following:

C++

```
virtual bool verify(const IceSSL::NativeConnectionInfoPtr&) = 0;
```

- The `getConnectionInfo` helper function has been removed because its functionality has been replaced by the `Connection::getInfo`operation. For example, in prior releases a C++ application would do the following:

C++

```
Ice::ConnectionPtr con = ...
IceSSL::ConnectionInfo info = IceSSL::getConnectionInfo(con);
```

Now the application should do this:

C++

```
Ice::ConnectionPtr con = ...
IceSSL::ConnectionInfoPtr info = IceSSL::ConnectionInfoPtr::dynamicCast(con->getInfo());
```

Alternatively, the application can downcast to the native class:

C++

```
Ice::ConnectionPtr con = ...
IceSSL::NativeConnectionInfoPtr info =
    IceSSL::NativeConnectionInfoPtr::dynamicCast(con->getInfo());
```

Migrating IceStorm and IceGrid databases from Ice 3.3

No changes were made to the database schema for IceStorm or IceGrid in this release. However, you still need to update your databases as described below.

Migrating Freeze databases from Ice 3.3

No changes were made that would affect the content of your Freeze databases. However, we upgraded the version of Berkeley DB, therefore when upgrading to Ice 3.4 you must also upgrade your database to the Berkeley DB 4.8 format. The only change that affects Freeze is the format of Berkeley DB's log file.

The instructions below assume that the database environment to be upgraded resides in a directory named `db` in the current working directory. For a more detailed discussion of database migration, please refer to the Berkeley DB Upgrade Process.

To migrate your database:

1. Shut down the old version of the application.
2. Make a backup copy of the database environment:

   ```
   > cp -r db backup.db      (Unix)
   > xcopy /E db backup.db   (Windows)
   ```

3. Locate the correct version of the Berkeley DB recovery tool (usually named `db_recover`). It is essential that you use the `db_recover` executable that matches the Berkeley DB version of your existing Ice release. For Ice 3.3, use `db_recover` from Berkeley DB 4.6. You can verify the version of your `db_recover` tool by running it with the `-V`option:

   ```
   > db_recover -V
   ```

4. Use the db_recover tool to run recovery on the database environment:

   ```
   > db_recover -h db
   ```

5. Recompile and install the new version of the application.
6. Force a checkpoint using the `db_checkpoint` utility. Note that you must use the `db_checkpoint`utility from Berkeley DB 4.8 when performing this step.

   ```
   > db_checkpoint -1 -h db
   ```

7. Restart the application.

Removed APIs in Ice 3.4.0

This section describes APIs that were deprecated in a previous release and have now been removed. Your application may no longer compile successfully if it relies on one of these APIs.

The following APIs were removed in Ice 3.4.0:

- `Glacier2.AddUserToAllowCategories`
  Use `Glacier2.Filter.Category.AcceptUser` instead.

- `Glacier2.AllowCategories`
  Use `Glacier2.Filter.Category.Accept` instead.

- `Ice.UseEventLog`
  Ice services (applications that use the C++ class `Ice::Service`) always use the Windows event log by default.

- `Communicator::setDefaultContext`
- `Communicator::getDefaultContext`
- `ObjectPrx:ice_defaultContext`
  Use the communicator's implicit request context instead.

- `nonmutating` keyword
  This keyword is no longer supported.

- `Freeze.UseNonmutating`
  Support for this property was removed along with the `nonmutating` keyword.

- `Ice::NegativeSizeException`
  The run time now throws `UnmarshalOutOfBoundsException` or `MarshalException` instead.

- `slice2docbook`
  This utility is no longer included in Ice.

- `Ice::AMD_Array_Object_ice_invoke`
  A new overloading of `ice_response` in the `AMD_Object_ice_invoke` class makes `AMD_Array_Object_ice_invoke` obsolete.

- Java2 mapping
  The Java2 mapping is no longer supported. Refer to Java language mapping changes in Ice 3.4 for more information.

Deprecated APIs in Ice 3.4.0

This section discusses APIs and components that are now deprecated. These APIs will be removed in a future Ice release, therefore we encourage you to update your applications and eliminate the use of these APIs as soon as possible.

The following APIs were deprecated in Ice 3.4.0:

- Asynchronous Method Invocation (AMI) interface
  The AMI interface in Ice 3.3 and earlier is now deprecated for C++, Java, and C#.

- `Glacier2.AddSSLContext`
  Replaced by `Glacier2.AddConnectionContext`.

- Standard platform methods should be used instead of the following:

  Java

  ```
  Ice.Object.ice_hash()         // Use hashCode
  Ice.ObjectPrx.ice_getHash()   // Use hashCode
  Ice.ObjectPrx.ice_toString()  // Use toString
  ```

  In Java, use `hashCode` and `toString`. In C#, use `GetHashCode` and `ToString`. In Ruby, use `hash` instead of `ice_getHash`.

- `Ice.Util.generateUUID()`
  In Java use `java.util.UUID.randomUUID().toString()`. In C# use `System.Guid.NewGuid.ToString()`.