Using a Freeze Map in Java

This section describes the Java code generator and demonstrates how to use a Freeze map in a Java program.

On this page:

`slice2freezej` Command Line Options

The Slice-to-Freeze compiler, `slice2freezej`, creates Java classes for Freeze maps. The compiler offers the following command-line options in addition to the standard options:

`--dict` *`NAME,KEY,VALUE`*

Generate a Freeze map class named *`NAME`* using *`KEY`* as key and *`VALUE`* as value. This option may be specified multiple times to generate several Freeze maps. *`NAME`* may be a scoped Java name, such as `Demo.Struct1ObjectMap`. *`KEY`* and *`VALUE`* represent Slice types and therefore must use Slice syntax, such as `bool` or `Ice::Identity`. The type identified by *`KEY`* must be a legal dictionary key type.

`--dict-index` *`MAP`*`[,`*`MEMBER`*`][,case-sensitive|case-insensitive]`

Add an index to the Freeze map named *`MAP`*. If *`MEMBER`* is specified, the map value type must be a structure or a class, and *`MEMBER`* must be the name of a member of that type. If *`MEMBER`* is not specified, the entire value is indexed. When the indexed member (or entire value) is a string, the index can be case-sensitive (default) or case-insensitive.

`--index` *`CLASS,TYPE,MEMBER`*`[,case-sensitive|case-insensitive]`

Generate an index class for a Freeze evictor. *`CLASS`* is the name of the index class to be generated. *`TYPE`* denotes the type of class to be indexed (objects of different classes are not included in this index). *`MEMBER`* is the name of the data member in *`TYPE`* to index. When *`MEMBER`* has type `string`, it is possible to specify whether the index is case-sensitive or not. The default is case-sensitive.

`--meta` *`META`*

Define the global metadata directive *`META`*. Using this option is equivalent to defining the global metadata *`META`* in each named Slice file, as well as in any file included by a named Slice file.

Generating a Simple Map for Java

As an example, the following command generates a simple map:

```
$ slice2freezej --dict StringIntMap,string,int
```

This command directs the compiler to create a map named `StringIntMap`, with the Slice key type `string` and the Slice value type `int`. The compiler produces one Java source file: `StringIntMap.java`.

`slice2freezej` Ant Task

In addition to the ant task for executing `slice2java`, Ice also includes an ant task for executing `slice2freezej`. The classes for `Slice2FreezeJTask` are stored in the same JAR file (`ant-ice.jar`) as `Slice2JavaTask`. Both tasks also share the same logic for locating a compiler in your execution environment and for managing dependencies between Slice files.

The `Slice2FreezeJTask` supports the parameters listed below:

Attribute	Description	Required
dependencyfile	Specifies an alternate name for the dependency file. If you specify a relative filename, it is relative to ant's current working directory. If not specified, the task uses the name `.depend` by default. If you do not define this attribute and `outputdir` is defined, the task creates the `.depend` file in the designated output directory (see `outputdir`).	No
ice	Instructs the Slice compiler to permit symbols that have the reserved prefix `Ice`. This parameter is used in the Ice build system and is not normally required by applications.	No
outputdir	Specifies the directory in which the Slice compiler generates Java source files. If not specified, the task uses ant's current working directory.	No
translator	Specifies the path name of the Slice compiler. If not specified, the task locates the Slice compiler in its execution environment as described for `slice2java`.	No

Several Slice compiler options must be defined as nested elements of the task:

- `define`
  Defines a preprocessor macro. The element supports the attributes `name` and (optionally) `value`, as shown below:

  XML

  ```
  <define name="FOO">
  <define name="BAR" value="5">
  ```

  These definitions are equivalent to the command-line options `-DFOO` and `-DBAR=5`, respectively.

- `dict`
  Generates a Freeze map. This element is equivalent to the `--dict` command line option and supports three attributes: `name`, `key`, and `value`.

- `dictindex`
  Generates an index for a Freeze map. This element is equivalent to the `--dict-index` command line option and supports three attributes: `name`, `member`, and `casesensitive`.

- `fileset`
  Specifies the set of Slice files to be compiled. Refer to the ant documentation of its `FileSet` type for more information.

- `includepath`
  Specifies the include file search path for Slice files. In ant terminology, `includepath` is a *path-like structure*. Refer to the ant documentation of its `Path` type for more information.

- `index`
  Generates an index for a Freeze evictor. This element is equivalent to the `--index` command line option and supports four attributes: `name`, `type`, `member`, and `casesensitive`.

- `meta`
  Defines a global metadata directive in each Slice file as well as in each included Slice file. The element supports the attributes `name` and `value`.

To enable the `Slice2FreezeJTask` in your ant project, define the following `taskdef` element in your project's build file:

XML

```
<taskdef name="slice2freezej" classname="Slice2FreezeJTask"/>
```

This configuration assumes that `ant-ice.jar` is already present in ant's class path. Alternatively, you can specify the JAR explicitly as follows:

XML

```
<taskdef name="slice2freezej" classpath="/opt/Ice/lib/ant-ice.jar" classname="Slice2FreezeJTask"/>
```

Once activated, you can invoke the task to translate your Slice files. The example shown below is a simplified version of the ant project for the `library` demo:

XML

```
<target name="generate" depends="init">
    <mkdir dir="generated"/>
    <slice2java outputdir="generated">
        <fileset dir="." includes="Library.ice"/>
    </slice2java>
    <slice2freezej ice="on" outputdir="generated">
        <fileset dir="/opt/Ice/slice/Ice" includes="BuiltinSequences.ice"/>
        <fileset dir="." includes="Library.ice"/>
        <dict name="StringIsbnSeqDict" key="string" value="Ice::StringSeq"/>
    </slice2freezej>
</target>
```

This invocation of the `slice2freezej` task enables the `ice` option because the generated Freeze map relies on a type that is defined in an Ice namespace and therefore loads the Slice file `BuiltinSequences.ice` directly.

The Freeze Map Class in Java

The class generated by `slice2freezej` implements the `Freeze.Map` interface, as shown below:

Java

```
package Freeze;

public interface Map<K, V> extends NavigableMap<K, V>
{
    void fastPut(K key, V value);
    void close();
    int closeAllIterators();
    void destroy();

    public interface EntryIterator<T> extends java.util.Iterator<T>
    {
        void close();
        void destroy(); // an alias for close
    }
}
```

The `Map` interface implements standard Java interfaces and provides nonstandard methods that improve efficiency and support database-oriented features. `Map` defines the following methods:

- `fastPut`
  Inserts a new key-value pair. This method is more efficient than the standard `put` method because it avoids the overhead of reading and decoding the previous value associated with the key (if any).
- `close`
  Closes the database associated with this map along with all open iterators. A map must be closed when it is no longer needed, either by closing the map directly or by closing the Freeze `Connection` object with which this map is associated.
- `closeAllIterators`
  Closes all open iterators and returns the number of iterators that were closed. We discuss iterators in more detail in the next section.
- `destroy`
  Removes the database associated with this map along with any indices.

`Map` inherits much of its functionality from the `Freeze.NavigableMap` interface, which derives from the standard Java interface `java.util.SortedMap` and also supports a subset of the `java.util.NavigableMap` interface from Java6:

Java

```
package Freeze;

public interface NavigableMap<K, V> extends java.util.SortedMap<K, V>
{
    java.util.Map.Entry<K, V> firstEntry();
    java.util.Map.Entry<K, V> lastEntry();

    java.util.Map.Entry<K, V> ceilingEntry(K key);
    java.util.Map.Entry<K, V> floorEntry(K key);
    java.util.Map.Entry<K, V> higherEntry(K key);
    java.util.Map.Entry<K, V> lowerEntry(K key);

    K ceilingKey(K key);
    K floorKey(K key);
    K higherKey(K key);
    K lowerKey(K key);

    java.util.Set<K> descendingKeySet();
    NavigableMap<K, V> descendingMap();

    NavigableMap<K, V> headMap(K toKey, boolean inclusive);
    NavigableMap<K, V> tailMap(K fromKey, boolean inclusive);
    NavigableMap<K, V> subMap(K fromKey, boolean fromInclusive,
                              K toKey, boolean toInclusive);

    java.util.Map.Entry<K, V> pollFirstEntry();
    java.util.Map.Entry<K, V> pollLastEntry();

    boolean fastRemove(K key);
}
```

The generated class does not implement `java.util.NavigableMap` because Freeze maps must remain compatible with Java5.

The `NavigableMap` interface provides a number of useful methods:

- `firstEntry`
  `lastEntry`
  Returns the first and last key-value pair, respectively.
- `ceilingEntry`
  Returns the key-value pair associated with the least key greater than or equal to the given key, or null if there is no such key.
- `floorEntry`
  Returns the key-value pair associated with the greatest key less than or equal to the given key, or null if there is no such key.
- `higherEntry`
  Returns the key-value pair associated with the least key greater than the given key, or null if there is no such key.
- `lowerEntry`
  Returns the key-value pair associated with the greatest key less than the given key, or null if there is no such key.
- `ceilingKey`
  `floorKey`
  `higherKey`
  `lowerKey`
  These methods have the same semantics as those described above, except they return only the key portion of the matching key-value pair or null if there is no such key.
- `descendingKeySet`
  Returns a set representing a reverse-order view of the keys in this map.
- `descendingMap`
  Returns a reverse-order view of the entries in this map.

- `headMap`
  Returns a view of the portion of this map whose keys are less than (or equal to, if inclusive is true) the given key.
- `tailMap`
  Returns a view of the portion of this map whose keys are greater than (or equal to, if inclusive is true) the given key.
- `subMap`
  Returns a view of the portion of this map whose keys are within the given range.
- `pollFirstEntry`
  `pollLastEntry`
  Removes and returns the first and last key-value pair, respectively.
- `fastRemove`
  Removes an existing key-value pair. As for `fastPut`, this method is a more efficient alternative to the standard `remove` method that returns true if a key-value pair was removed, or false if no match was found.

> You must supply a comparator object when constructing the map in order to use many of these methods.

Note that `NavigableMap` also inherits overloaded methods named `headMap`, `tailMap`, and `subMap` from the `SortedMap` interface. These methods have the same semantics as the ones defined in `NavigableMap` but they omit the boolean arguments (refer to the JDK documentation for complete details). Although these methods are declared as returning a `SortedMap`, the actual type of the returned object is a `NavigableMap` that you can downcast if necessary.

There are some limitations in the sub maps returned by the `headMap`, `tailMap` and `subMap` methods:

- A new entry in the Freeze map cannot be added via a sub map, therefore calling `put` raises `UnsupportedOperationException`.
- An existing entry in the Freeze map cannot be removed via a sub map or iterator for a secondary key.

Now let us examine the contents of the source file created by the example in the previous section:

Java

```
public class StringIntMap extends ...
    // implements Freeze.Map<String, Integer>
{
    public StringIntMap(
        Freeze.Connection connection,
        String dbName,
        boolean createDb,
        java.util.Comparator<String> comparator);

    public StringIntMap(
        Freeze.Connection connection,
        String dbName,
        boolean createDb);

    public StringIntMap(
        Freeze.Connection connection,
        String dbName);
}
```

`StringIntMap` derives from an internal Freeze base class that implements the interface `Freeze.Map<String, Integer>`. The generated class defines several overloaded constructors whose arguments are described below:

- `connection`
  The Freeze connection object.
- `dbName`
  The name of the database in which to store this map's persistent state. Note that a database can only contain the persistent state of one map type. Any attempt to instantiate maps of different types on the same database results in undefined behavior.
- `createDb`
  A flag indicating whether the map should create the database if it does not already exist. If this argument is not specified, the default value is `true`.
- `comparator`
  An object used to compare the map's keys. If this argument is not specified, the default behavior compares the encoded form of the keys.

Why Comparators are Important

The constructor of a Freeze map optionally accepts a comparator object for the primary key and, if any indices are generated, a second object that supplies comparators for each of the index keys. If you do not supply a comparator, Freeze simply compares the encoded form of the keys. This default behavior is acceptable when comparing keys for equality, but using the encoded form cannot work reliably when comparing keys for ordering purposes.

For example, many of the methods in `NavigableMap` perform greater-than or less-than comparisons on keys, including `ceilingEntry`, `headMap`, and `tailMapFor`*`MEMBER`*. All of these methods raise `UnsupportedOperationException` if you failed to supply a corresponding comparator when constructing the map. (The same applies to `NavigableMap` objects created for secondary keys.) In fact, the only `NavigableMap` methods that do *not* require a comparator are `firstEntry`, `lastEntry`, `pollFirstEntry`, `pollLastEntry`, and `fastRemove`.

As you can see, the functionality of a Freeze map is quite limited if no comparators are configured, therefore we recommend using comparators at all times.

Using Iterators with Freeze Maps in Java

You can iterate over a Freeze map just as you can with any container that implements the `java.util.Map` interface. For example, the code below displays the key and value of each element:

Java

```
StringIntMap m = new StringIntMap(...);
java.util.Iterator<java.util.Map.Entry<String, Integer>> i = m.entrySet().iterator();
while (i.hasNext()) {
    java.util.Map.Entry<String, Integer> e = i.next();
    System.out.println("Key: " + e.getKey());
    System.out.println("Value: " + e.getValue());
}
```

Generally speaking, a program should close an iterator when it is no longer necessary. (An iterator that is garbage collected without being closed emits a warning message.) However, an explicit close was not necessary in the preceding example because Freeze automatically closes a read-only iterator when it reaches the last element (a read-only iterator is one that is opened outside of any transaction). If instead our program had stopped using the iterator prior to reaching the last element, an explicit close would have been necessary:

Java

```
StringIntMap m = new StringIntMap(...);
java.util.Iterator<java.util.Map.Entry<String, Integer>> i = m.entrySet().iterator();
while (i.hasNext()) {
    java.util.Map.Entry<String, Integer> e = i.next();
    System.out.println("Key: " + e.getKey());
    System.out.println("Value: " + e.getValue());
    if (e.getValue().intValue() == 5)
        break;
}
((Freeze.Map.EntryIterator)i).close();
```

Closing the iterator requires downcasting it to a Freeze-specific interface named `Freeze.Map.EntryIterator`. The definition of this interface was shown in the previous section.

Freeze maps also support the enhanced `for` loop functionality in Java5. Here is a simpler way to write our original program:

Java

```
StringIntMap m = new StringIntMap(...);
for (java.util.Map.Entry<String, Integer> e : m.entrySet()) {
    System.out.println("Key: " + e.getKey());
    System.out.println("Value: " + e.getValue());
}
```

As in the first example, Freeze automatically closes the iterator when no more elements are available. Although the enhanced `for` loop is convenient, it is not appropriate for all situations because the loop hides its iterator and therefore prevents the program from accessing the iterator in order to close it. In this case, you can use the traditional `while` loop instead of the `for` loop, or you can invoke `closeAllIterators` on the map as shown below:

Java

```
StringIntMap m = new StringIntMap(...);
for (java.util.Map.Entry<String, Integer> e : m.entrySet()) {
    System.out.println("Key: " + e.getKey());
    System.out.println("Value: " + e.getValue());
    if (e.getValue().intValue() == 5)
        break;
}
int num = m.closeAllIterators();
assert(num <= 1); // The iterator may already be closed.
```

The `closeAllIterators` method returns an integer representing the number of iterators that were actually closed. This value can be useful for diagnostic purposes, such as to assert that a program is correctly closing its iterators.

Generating Indices for Freeze Maps in Java

Using the `--dict-index` option to define an index for a secondary key causes `slice2freezej` to generate the following additional code in a Freeze map:

- A static nested class named `IndexComparators`, which allows you to supply a custom comparator object for each index in the map.
- An overloading of the map constructor that accepts an instance of `IndexComparators`.
- An overloading of the `recreate` method that accepts an instance of `IndexComparators`.
- Searching, counting, and range-searching methods for finding key-value pairs using the secondary key.

We discuss each of these additions in more detail below. In this discussion, *`MEMBER`* refers to the optional argument of the `--dict-index` option, and *`MEMBER_TYPE`* refers to the type of that member. As explained earlier, if *`MEMBER`* is not specified, `slice2freezej` creates an index for the value type of the map. The sample code presented in this section assumes we have generated a Freeze map using the following command:

Java

```
$ slice2freezej --dict StringIntMap,string,int --dict-index StringIntMap
```

By default, index keys are sorted using their binary Ice-encoded representation. This is an efficient sorting scheme but does not necessarily provide a meaningful traversal order for applications. You can choose a different order by providing an instance of the `IndexComparators` class to the map constructor. This class has a public data member holding a comparator (an instance of `java.util.Comparator<`*`MEMBER_TYPE`*`>`) for each index in the map. The class also provides an empty constructor as well as a convenience constructor that allows you to instantiate and initialize the object all at once. The name of each data member is *`MEMBER`*`Comparator`. If *`MEMBER`* is not specified, the `IndexComparators` class has a single data member named `valueComparator`.

> Much of the functionality offered by a map index requires that you provide a custom comparator.

Here is the definition of `IndexComparators` for `StringIntMap`:

Java

```
public class StringIntMap ... {
    public static class IndexComparators {
        public IndexComparators() {}

        public IndexComparators(java.util.Comparator<Integer> valueComparator);

        public java.util.Comparator<Integer> valueComparator;
    }

    ...
}
```

To instantiate a Freeze map using your custom comparators, you must use the overloaded constructor that accepts the `IndexComparators` object. For our `StringIntMap`, this constructor has the following definition:

Java

```
public class StringIntMap ... {
    public StringIntMap(
        Freeze.Connection connection,
        String dbName,
        boolean createDb,
        java.util.Comparator<String> comparator,
        IndexComparators indexComparators);

    ...
}
```

Now we can instantiate our `StringIntMap` as follows:

Java

```
java.util.Comparator<String> myMainKeyComparator = ...;
StringIntMap.IndexComparators indexComparators = new StringIntMap.IndexComparators();
indexComparators.valueComparator = ...;
StringIntMap m = new StringIntMap(connection, "stringIntMap", true,
                                  myMainKeyComparator, indexComparators);
```

If you later need to change the index configuration of a Freeze map, you can use one of the `recreate` methods to update the database. Here are the definitions from `StringIntMap`:

Java

```
public class StringIntMap ... {
    public static void recreate(
        Freeze.Connection connection,
        String dbName,
        java.util.Comparator<String> comparator);

    public static void recreate(
        Freeze.Connection connection,
        String dbName,
        java.util.Comparator<String> comparator,
        IndexComparators indexComparators);

    ...
}
```

The first overloading is generated for every map, whereas the second overloading is only generated when the map has at least one index. As its name implies, the `recreate` method creates a new copy of the database. More specifically, the method removes any existing indices, copies every key-value pair to a temporary database, and finally replaces the old database with the new one. As a side-effect, this process also populates any remaining indices. The first overloading of `recreate` is useful when you have regenerated the map to remove the last index and wish to clean up the map's database state.

`slice2freezej` also generates a number of index-specific methods. The names of these methods incorporate the member name (*`MEMBER`*), or use `value` if *`MEMBER`* is not specified. In each method name, the value of *`MEMBER`* is used unchanged if it appears at the beginning of the method's name. Otherwise, if *`MEMBER`* is used elsewhere in the method name, its first letter is capitalized. The index methods are described below:

- `public Freeze.Map.EntryIterator<Map.Entry<K, V>>`
  `findBy`*`MEMBER`*`(`*`MEMBER_TYPE`* `index)`

  `public Freeze.Map.EntryIterator<Map.Entry<K, V>>`
  `findBy`*`MEMBER`*`(`*`MEMBER_TYPE`* `index, boolean onlyDups)`
  Returns an iterator over elements of the Freeze map starting with an element with whose index value matches the given index value. If there is no such element, the returned iterator is empty (`hasNext` always returns false). When the second parameter is true (or is not provided), the returned iterator provides only "duplicate" elements, that is, elements with the very same index value. Otherwise, the iterator sets a starting position in the map, and then provides elements until the end of the map, sorted according to the index comparator. Any attempt to modify the map via this iterator results in an `UnsupportedOperationException`.

- `public int` *`MEMBER`*`Count(`*`MEMBER_TYPE`* `index)`
  Returns the number of elements in the Freeze map whose index value matches the given index value.

- `public NavigableMap<`*`MEMBER_TYPE`*`, Set<Map.Entry<K, V>>>`
  `headMapFor`*`MEMBER`*`(`*`MEMBER_TYPE`* `to, boolean inclusive)`

  `public NavigableMap<`*`MEMBER_TYPE`*`, Set<Map.Entry<K, V>>>`
  `headMapFor`*`MEMBER`*`(`*`MEMBER_TYPE`* `to)`
  Returns a view of the portion of the Freeze map whose keys are less than (or equal to, if `inclusive` is true) the given key. If `inclusive` is not specified, the method behaves as if `inclusive` is false.

- `public NavigableMap<`*`MEMBER_TYPE`*`, Set<Map.Entry<K, V>>>`
  `tailMapFor`*`MEMBER`*`(`*`MEMBER_TYPE`* `from, boolean inclusive)`

- public NavigableMap<*`MEMBER_TYPE`*, Set<Map.Entry<K, V>>>}}
  `tailMapFor`*`MEMBER`*`(`*`MEMBER_TYPE`* `from)`
  Returns a view of the portion of the Freeze map whose keys are greater than (or equal to, if `inclusive` is true) the given key. If `inclusive` is not specified, the method behaves as if `inclusive` is true.

- `public NavigableMap<`*`MEMBER_TYPE`*`, Set<Map.Entry<K, V>>>`
  `subMapFor`*`MEMBER`*`(`*`MEMBER_TYPE`* `from, boolean fromInclusive,`
  `MEMBER_TYPE` `to, boolean toInclusive)`

  `public NavigableMap<`*`MEMBER_TYPE`*`, Set<Map.Entry<K, V>>>`
  `subMapFor`*`MEMBER`*`(`*`MEMBER_TYPE`* `from, `*`MEMBER_TYPE`* `to)`
  Returns a view of the portion of the Freeze map whose keys are within the given range. If `fromInclusive` and `toInclusive` are not specified, the method behaves as if `fromInclusive` is true and `toInclusive` is false.

- `public NavigableMap<`*`MEMBER_TYPE`*`, Set<Map.Entry<K, V>>>`
  `mapFor`*`MEMBER`*`()`
  Returns a view of the entire Freeze map ordered by the index key.

For the methods returning a `NavigableMap`, the key type is the secondary key type and the value is the set of matching key-value pairs from the Freeze map. (For the sake of readability, we have omitted the `java.util` prefix from `Set` and `Map.Entry`.) In other words, the returned map is a mapping of the secondary key to all of the entries whose value contains the same key. Any attempt to add, remove, or modify an element via a sub map view or an iterator of a sub map view results in an `UnsupportedOperationException`.

Note that iterators returned by the `findBy`*`MEMBER`* methods, as well as those created for sub map views, may need to be closed explicitly, just like iterators obtained for the main Freeze map.

Here are the definitions of the index methods for `StringIntMap`:

Java

```
public Freeze.Map.EntryIterator<Map.Entry<String, Integer>>
findByValue(Integer index);

public Freeze.Map.EntryIterator<Map.Entry<String, Integer>>
findByValue(Integer index, boolean onlyDups);

public int valueCount(Integer index);

public NavigableMap<Integer, Set<Map.Entry<String, Integer>>>
headMapForValue(Integer to, boolean inclusive);
public NavigableMap<Integer, Set<Map.Entry<String, Integer>>>
headMapForValue(Integer to);

public NavigableMap<Integer, Set<Map.Entry<String, Integer>>>
tailMapForValue(Integer from, boolean inclusive);
public NavigableMap<Integer, Set<Map.Entry<String, Integer>>>
tailMapForValue(Integer from);

public NavigableMap<Integer, Set<Map.Entry<String, Integer>>>
subMapForValue(Integer from, boolean fromInclusive,
               Integer to, boolean toInclusive);
public NavigableMap<Integer, Set<Map.Entry<String, Integer>>>
subMapForValue(Integer from, Integer to);

public NavigableMap<Integer, Set<Map.Entry<String, Integer>>>
mapForValue();
```

Sample Freeze Map Program in Java

The program below demonstrates how to use a `StringIntMap` to store `<string, int>` pairs in a database. You will notice that there are no explicit `read` or `write` operations called by the program; instead, simply using the map has the side effect of accessing the database.

Java

```
public class Client
{
    public static void
    main(String[] args)
    {
        // Initialize the Communicator.
        //
        Ice.Communicator communicator = Ice.Util.initialize(args);

        // Create a Freeze database connection.
        //
        Freeze.Connection connection = Freeze.Util.createConnection(communicator, "db");

        // Instantiate the map.
        //
        StringIntMap map = new StringIntMap(connection, "simple", true);

        // Clear the map.
        //
        map.clear();

        int i;

        // Populate the map.
        //
        for (i = 0; i < 26; i++) {
            final char[] ch = { (char)('a' + i) };
            map.put(new String(ch), i);
        }

        // Iterate over the map and change the values.
        //
        for (java.util.Map.Entry<String, Integer> e : map.entrySet()) {
            Integer in = e.getValue();
            e.setValue(in.intValue() + 1);
        }

        // Find and erase the last element.
        //
        boolean b;
        b = map.containsKey("z");
        assert(b);
        b = map.fastRemove("z");
        assert(b);

        // Clean up.
        //
        map.close();
        connection.close();
        communicator.destroy();

        System.exit(0);
    }
}
```

Prior to instantiating a Freeze map, the application must connect to a Berkeley DB database environment:

Java

```
Freeze.Connection connection = Freeze.Util.createConnection(communicator, "db");
```

The second argument is the name of a Berkeley DB database environment; by default, this is also the file system directory in which Berkeley DB creates all database and administrative files.

Next, the code instantiates the `StringIntMap` on the connection. The constructor's second argument supplies the name of the database file, and the third argument indicates that the database should be created if it does not exist:

Java

```
StringIntMap map = new StringIntMap(connection, "simple", true);
```

After instantiating the map, we clear it to make sure it is empty in case the program is run more than once:

Java

```
map.clear();
```

We populate the map, using a single-character string as the key. As with `java.util.Map`, the key and value types must be Java objects but the compiler takes care of autoboxing the integer argument:

Java

```
for (i = 0; i < 26; i++) {
    final char[] ch = { (char)('a' + i) };
    map.put(new String(ch), i);
}
```

Iterating over the map is no different from iterating over any other map that implements the `java.util.Map` interface:

Java

```
for (java.util.Map.Entry<String, Integer> e : map.entrySet()) {
    Integer in = e.getValue();
    e.setValue(in.intValue() + 1);
}
```

Next, the program verifies that an element exists with key `z`, and then removes it using `fastRemove`:

Java

```
b = map.containsKey("z");
assert(b);
b = map.fastRemove("z");
assert(b);
```

Finally, the program closes the map and its connection.

Java

```
map.close();
connection.close();
```

See Also

- Using the Slice Compilers
- slice2java Command-Line Options
- Freeze Map Concepts