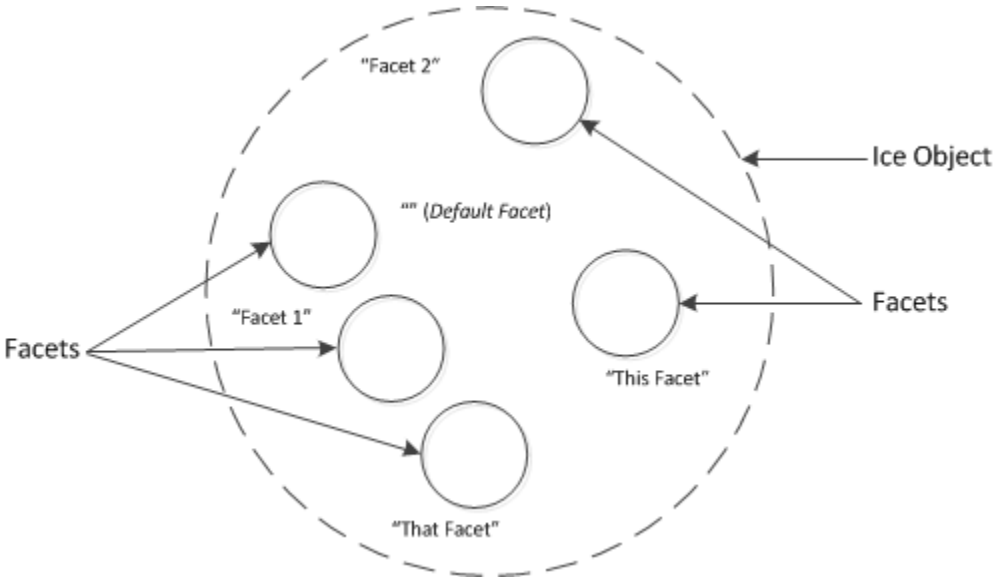
Facet Concepts

On this page:

Ice Objects as Collections of Facets

Up to this point, we have presented an Ice object as a single conceptual entity, that is, as an object with a single most-derived interface and a single id entity, with the object being implemented by a single servant. However, an Ice object is more correctly viewed as a collection of one or more sub-objects known as facets, as shown below:



An Ice object with five facets sharing a single object identity.

The diagram above shows a single Ice object with five facets. Each facet has a name, known as the *facet name*. Within a single Ice object, all facets must have unique names. Facet names are arbitrary strings that are assigned by the server that implements an Ice object. A facet with an empty facet name is legal and known as the *default facet*. Unless you arrange otherwise, an Ice object has a single default facet; by default, operations that involve Ice objects and servants operate on the default facet.

Note that all the facets of an Ice object share the same single identity, but have different facet names. Recall the definition of `Ice::Current` once more:

Slice

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

    enum OperationMode { Normal, \Nonmutating, \Idempotent };

    local struct Current {
        ObjectAdapter   adapter;
        Identity        id;
        string          facet;
        string          operation;
        OperationMode   mode;
        Context         ctx;
        int             requestId;
    };
};
```

By definition, if two facets have the same `id` field, they are part of the same Ice object. Also by definition, if two facets have the same `id` field, their `facet` fields have different values.

Even though Ice objects usually consist of just the default facet, it is entirely legal for an Ice object to consist of facets that all have non-empty names (that is, it is legal for an Ice object not to have a default facet).

Each facet has a single most-derived interface. There is no need for the interface types of the facets of an Ice object to be unique. It is legal for two facets of an Ice object to implement the same most-derived interface.

Each facet is implemented by a servant. All the usual implementation techniques for servants are available to implement facets — for example, you can implement a facet using a servant locator. Typically, each facet of an Ice object has a separate servant, although, if two facets of an Ice object have the same type, they can also be implemented by a single servant (for example, using a default servant).

Server-Side Facet Operations

On the server side, the object adapter offers a number of operations to support facets:

Slice

```
namespace Ice {
  dictionary<string, Object> FacetMap;

  local interface ObjectAdapter {
     Object*  addFacet(Object servant, Identity id, string facet);
     Object*  addFacetWithUUID(Object servant, string facet);
     Object   removeFacet(Identity id, string facet);
     Object   findFacet(Identity id, string facet);

     FacetMap findAllFacets(Identity id);
     FacetMap removeAllFacets(Identity id);
     // ...
  };
};
```

These operations have the same semantics as the corresponding "normal" operations for servant activation and deactivation (`add`, `addWithUUID`, `remove`, and `find`), but also accept a facet name. The corresponding "normal" operations are simply convenience operations that supply an empty facet name. For example, `remove(id)` is equivalent to `removeFacet(id, "")`, that is, `remove(id)` operates on the default facet.

`findAllFacets` returns a dictionary of <*`facet-name, servant`*> pairs that contains all the facets for the given identity.

`removeAllFacets` removes all facets for a given identity from the active servant map, that is, it removes the corresponding Ice object entirely. The operation returns a dictionary of <*`facet-name, servant`*> pairs that contains all the removed facets.

These operations are sufficient for the server to create Ice objects with any number of facets. For example, assume that we have the following Slice definitions:

Slice

```
module Filesystem {
    // ...

    interface File extends Node {
        idempotent Lines read();
        idempotent void write(Lines text) throws GenericError;
    };
};

module FilesystemExtensions {
    // ...

    class DateTime extends TimeOfDay {
        // ...
    };

    struct Times {
        DateTime createdDate;
        DateTime accessedDate;
        DateTime modifiedDate;
    };

    interface Stat {
        idempotent Times getTimes();
    };
};
```

Here, we have a `File` interface that provides operations to read and write a file, and a `Stat` interface that provides access to the file creation, access, and modification time. (Note that the `Stat` interface is defined in a different module and could also be defined in a different source file.) If the server wants to create an Ice object that contains a `File` instance as the default facet and a `Stat` instance that provides access to the time details of the file, it could do so as follows:

C++

```
// Create a File instance.
//
Filesystem::FilePtr file = new FileI;

// Create a Stat instance.
//
FilesystemExctensions::DateTimePtr dt = new FilesystemExtensions::DateTime;
FilesystemExtensions::Times times;
times.createdDate = dt;
times.accessedDate = dt;
times.modifiedDate = dt;
FilesystemExtensions::StatPtr stat = new StatI(times);

// Register the File instance as the default facet.
//
Filesystem::FilePrx filePrx = myAdapter->addWithUUID(file);

// Register the Stat instance as a facet with name "Stat".
//
myAdapter->addFacet(stat, filePrx->ice_getIdentity(), "Stat");
```

The first few lines simply create and initialize a `FileI` and `StatI` instance. (The details of this do not matter here.) All the action is in the last two statements:

C++

```
Filesystem::FilePrx filePrx = myAdapter->addWithUUID(file);
myAdapter->addFacet(stat, filePrx->ice_getIdentity(), "Stat");
```

This registers the `FileI` instance with the object adapter as usual. (In this case, we let the Ice run time generate a UUID as the object identity.) Because we are calling `addWithUUID` (as opposed to `addFacetWithUUID`), the instance becomes the default facet.

The second line adds a facet to the instance with the facet name `Stat`. Note that we call `ice_getIdentity` on the `File` proxy to pass an object identity to `addFacet`. This guarantees that the two facets share the same object identity.

Note that, in general, it is a good idea to use `ice_getIdentity` to obtain the identity of an existing facet when adding a new facet. That way, it is guaranteed that the facets share the same identity. (If you accidentally pass a different identity to `addFacet`, you will not add a facet to an existing Ice object, but instead register a new Ice object; using `ice_getIdentity` makes this mistake impossible.)

Client-Side Facet Operations

On the client side, which facet a request is addressed to is implicit in the proxy that is used to send the request. For an application that does not use facets, the facet name is always empty so, by default, requests are sent to the default facet.

The client can use a `checkedCast` to obtain a proxy for a particular facet. For example, assume that the client obtains a proxy to a `File` instance as shown above. The client can cast between the `File` facet and the `Stat` facet (and back) as follows:

C++

```
// Get a File proxy.
//
Filesystem::FilePrx file = ...;

// Get the Stat facet.
//
FilesystemExtensions::StatPrx stat =
    FilesystemExtensions::StatPrx::checkedCast(file, "Stat");

// Go back from the Stat facet to the File facet.
//
Filesystem::FilePrx file2 = Filesystem::FilePrx::checkedCast(stat, "");

assert(file2 == file); // The two proxies are identical.
```

This example illustrates that, given any facet of an Ice object, the client can navigate to any other facet by using a `checkedCast` with the facet name.

If an Ice object does not provide the specified facet, `checkedCast` returns null:

C++

```
FilesystemExtensions::StatPrx stat =
    FilesystemExtensions::StatPrx::checkedCast(file, "Stat");

if (!stat) {
    // No Stat facet on this object, handle error...
} else {
    FilesystemExtensions::Times times = stat->getTimes();

    // Use times struct...
}
```

Note that `checkedCast` also returns a null proxy if a facet exists, but the cast is to the wrong type. For example:

C++

```
// Get a File proxy.
//
Filesystem::FilePrx file = ...;

// Cast to the wrong type.
//
SomeTypePrx prx = SomeTypePrx::checkedCast(file, "Stat");

assert(!prx); // checkedCast returns a null proxy.
```

If you want to distinguish between non-existence of a facet and the facet being of the incorrect type, you can first obtain the facet as type `Object` and then down-cast to the correct type:

C++

```
// Get a File proxy.
//
Filesystem::FilePrx file = ...;

// Get the facet as type Object.
//
Ice::ObjectPrx obj = Ice::ObjectPrx::checkedCast(file, "Stat");
if (!obj) {
    // No facet with name "Stat" on this Ice object.
} else {
    FilesystemExtensions::StatPrx stat =
        FilesystemExtensions::StatPrx::checkedCast(file);
    if (!stat) {
        // There is a facet with name "Stat", but it is not
        // of type FilesystemExtensions::Stat.
    } else {
        // Use stat...
    }
}
```

This last example also illustrates that

C++

```
StatPrx::checkedCast(prx, "")
```

is *not* the same as

C++

```
StatPrx::checkedCast(prx)
```

The first version explicitly requests a cast to the default facet. This means that the Ice run time first looks for a facet with the empty name and then attempts to down-cast that facet (if it exists) to the type `Stat`.

The second version requests a down-cast that *preserves* whatever facet is currently effective in the proxy. For example, if the `prx` proxy currently holds the facet name "Joe", then (if `prx` points at an object of type `Stat`) the run time returns a proxy of type `StatPrx` that also stores the facet name "Joe".

It follows that, to navigate between facets, you must always use the two-argument version of `checkedCast`, whereas, to down-cast to another type while preserving the facet name, you must always use the single-argument version of `checkedCast`.

You can always check what the current facet of a proxy is by calling `ice_getFacet`:

C++

```
Ice::ObjectPrx obj = ...;

cout << obj->ice_getFacet() << endl; // Print facet name
```

This prints the facet name. (For the default facet, `ice_getFacet` returns the empty string.)

Facet Exception Semantics

The common exceptions `ObjectNotExistException` and `FacetNotExistException` have the following semantics:

- `ObjectNotExistException`
  This exception is raised only if no facets exist at all for a given object identity.
- `FacetNotExistException`
  This exception is raised only if at least one facet exists for a given object identity, but not the specific facet that is the target of an operation invocation.

If you are using servant locators or default servants, you must take care to preserve these semantics. In particular, if you return null from a servant locator's `locate` operation, this appears to the client as an `ObjectNotExistException`. If the object identity for a request is known (that is, there is at least one facet with that identity), but no facet with the specified name exists, you must explicitly throw a `FacetNotExistException` from `locate` instead of simply returning null.

See Also

- Object Identity
- Run-Time Exceptions
- Object Adapters
- Servant Locators
- Default Servants
- The Current Object