


Object Creation

Now that we understand what it means for an Ice object to exist, we can look at what is involved in creating an Ice object. Fundamentally, there is only one way for an Ice object to come into being: the server must instantiate a servant for the object and add an entry for that servant to the [Active Servant Map](#) (ASM) (or, alternatively, arrange for a [servant locator](#) to return a servant from its `locate` operation).

 For the remainder of this chapter, we will ignore the distinction between using the ASM and a servant locator and simply assume that the code uses the ASM. This is because servant locators do not alter the discussion: if `locate` returns a servant, that is the same as a successful lookup in the ASM; if `locate` returns null or throws `ObjectNotExistException`, that is the same as an unsuccessful lookup in the ASM.

One obvious way for a server to create a servant is to, well, simply instantiate it and add it to the ASM of its own accord. For example:

C++

```
DirectoryIPtr root = new DirectoryI("/", 0);
adapter?>addWithUUID(root); // Ice object exists now
```

The servant exists as soon as the call to `new` completes, and the Ice object exists as soon as the code adds the servant to the ASM: at that point, the Ice object becomes reachable to clients who hold a proxy to it.

This is the way we created Ice objects for our file system application earlier in the manual. However, doing so is not all that interesting because the only files and directories that exist are those that the server decides to create when it starts up. What we really want is a way for *clients* to create and destroy directories and files.

On this page:

- [Creating an Object with a Factory](#)
- [Implementing a Factory Operation](#)

Creating an Object with a Factory


The canonical way to create an object is to use the factory pattern [\[1\]](#). The factory pattern, in a nutshell, says that objects are created by invoking an operation (usually called `create`) on an object factory:

Slice

```
interface PhoneEntry {
    idempotent string name();
    idempotent string getNumber();
    idempotent void setNumber(string phNum);
};

exception PhoneEntryExists {
    string name;
    string phNum;
};

interface PhoneEntryFactory {
    PhoneEntry* create(string name, string phNum)
        throws PhoneEntryExists;
};
```

 Rather than continue with the file system example, we will simplify the discussion for the time being by using the phone book example mentioned earlier; we will return to the file system application to explore [more complex issues](#).

The entries in the phone book consist of simple name-number pairs. The interface to each entry is called `PhoneEntry` and provides operations to read the name and to read and write the phone number. (For a real application, the objects would likely be more complex and encapsulate more state. However, these simple objects will do for the purposes of this discussion.)

To create a new entry, a client calls the `create` operation on a `PhoneEntryFactory` object. (The factory is a singleton object [1] — that is, only one instance of that interface exists in the server.) It is the job of `create` to create a new `PhoneEntry` object, using the supplied name as the [object identity](#).

An immediate consequence of using the name as the object identity is that `create` can raise a `PhoneEntryExists` exception: presumably, if a client attempts to create an entry with the same name as an already-existing entry, we need to let the client know about this. (Whether this is an appropriate design is something we examine more closely in [Object Identity and Uniqueness](#).)

`create` returns a proxy to the newly-created object, so the client can use that proxy to invoke operations. However, this is by convention only. For example, `create` could be a `void` operation if the client has some other way to eventually get a proxy to the new object (such as creating the proxy from a string, or locating the proxy via a search operation). Alternatively, you could define "bulk" creation operations that allow clients to create several new objects with a single RPC. As far as the Ice run time is concerned, there is nothing special about a factory operation: a factory operation is just like any other operation; it just so happens that a factory operation creates a new Ice object as a side effect of being called, that is, the *implementation* of the operation is what creates the object, not the Ice run time.

Also note that `create` accepts a name and a `phNum` parameter, so it can initialize the new object. This is not compulsory, but generally a good idea. An alternate factory operation could be:

Slice

```
interface PhoneEntryFactory {
    PhoneEntry* create(string name)
        throws PhoneEntryExists;
};
```

With this design, the assumption is that the client will call `setNumber` after it has created the object. However, in general, allowing objects that are not fully initialized is a bad idea: it all too easily happens that a client either forgets to complete the initialization, or happens to crash or get disconnected before it can complete the initialization. Either way, we end up with a partially-initialized object in the system that can cause surprises later.



This is the approach taken by COM's `CoCreateObject`, which suffers from just that problem.

Similarly, so-called `generic` factories are also something to be avoided:

Slice

```
dictionary<string, string> Params;

exception CannotCreateException {
    string reason;
};

interface GenericFactory {
    Object* create(Params p)
        throws CannotCreateException;
};
```

The intent here is that a `GenericFactory` can be used to create any kind of object; the `Params` dictionary allows an arbitrary number of parameters to be passed to the `create` operation in the form of name — value pairs, for example:

C++

```
GenericFactoryPrx factory = ...;

Ice::ObjectPrx obj;
Params p;

// Make a car.
//
p["Make"] = "Ford";
p["Model"] = "Falcon";
obj = factory?>create(p);
CarPrx car = CarPrx::checkedCast(obj);

// Make a horse.
//
p.clear();
p["Breed"] = "Clydesdale";
p["Sex"] = "Male";
obj = factory?>create(p);
HorsePrx horse = HorsePrx::checkedCast(obj);
```

We strongly discourage you from creating factory interfaces such as this, unless you have a good overriding reason: generic factories undermine type safety and are much more error-prone than strongly-typed factories.

Implementing a Factory Operation

The implementation of an object factory is simplicity itself. Here is how we could implement the `create` operation for our `PhoneEntryFactory`:

C++

```
PhoneEntryPrx
PhoneEntryFactory::create(const string& name, const string& phNum, const Current& c)
{
    try {
        CommunicatorPtr comm = c.adapter.getCommunicator();
        PhoneEntryPtr servant = new PhoneEntryI(name, phNum);
        return PhoneEntryPrx::uncheckedCast(c.adapter?>add(servant, comm?>stringToIdentity(name)));
    } catch (const Ice::AlreadyRegisteredException&) {
        throw PhoneEntryExists(name, phNum);
    }
}
```

The `create` function instantiates a new `PhoneEntryI` object (which is the servant for the new `PhoneEntry` object), adds the servant to the ASM, and returns the proxy for the new object. Adding the servant to the ASM is what creates the new Ice object, and client requests are dispatched to the new object as soon as that entry appears in the ASM (assuming the [object adapter is active](#)).

Note that, even though this code contains no explicit lock, it is thread-safe. The `add` operation on the object adapter is atomic: if two clients concurrently add a servant with the same identity, exactly one thread succeeds in adding the entry to the ASM; the other thread receives an `AlreadyRegisteredException`. Similarly, if two clients concurrently call `create` for different entries, the two calls execute concurrently in the server (if the server is multi-threaded); the implementation of `add` in the Ice run time uses appropriate locks to ensure that concurrent updates to the ASM cannot corrupt anything.

See Also

- [The Active Servant Map](#)
- [Servant Locators](#)
- [Object Identity](#)
- [Object Identity and Uniqueness](#)
- [Object Life Cycle for the File System Application](#)
- [Object Adapter States](#)
- [Servant Activation and Deactivation](#)

References

1. Gamma, E., et al. 1994. [Design Patterns](#). Reading, MA: Addison-Wesley.