

# Optional Values



Version 1.1 of the Ice [encoding](#) introduces a convenient new feature that allows you to declare data members and parameters as optional. Collectively called *optional values*, this feature provides two main benefits for Ice applications:

- It offers a simple and natively-supported mechanism for representing values that may or may not be present, without consuming any additional space in a message when an optional value is not used.
- It significantly improves the ability to gradually evolve components of an Ice application while maintaining backward compatibility with existing deployments.

On this page:

- [Overview of Optional Values](#)
  - [Optional Parameters in Operations](#)
  - [Optional Data Members in Classes and Exceptions](#)
- [Guidelines for using Optional Values](#)
- [Backward Compatibility with Encoding Version 1.0](#)
- [Using Optional Values Efficiently](#)

## Overview of Optional Values

The optional syntax is the same for both parameters and data members:

```
optional(tag) type name
```

The `optional` keyword denotes an optional value with the given tag. The value for `tag` must be a non-negative integer.



If your Slice definitions currently use `optional` as an identifier, you can continue using it as an identifier by escaping it as `\optional`.

Data members and parameters not marked as optional are considered *required*; a sender must supply values for all required parameters and members.

Each language mapping defines APIs for passing an optional value, indicating that an optional value is not set, and testing whether an optional value is set. For example, the three strongly-typed languages that Ice currently supports (C++, C#, and Java) all use a similar API: an `Optional` type wraps the optional value and provides methods for examining and changing its value. Refer to the language mapping sections for more details on optional values.



A well-behaved program must not assume that an optional data member or parameter always has a value.

An optional value incurs no additional overhead in the encoding when its value is unset. If an optional parameter or member has a value, its overhead depends on [several factors](#), including your choice of tag values.

[Back to Top ^](#)

## Optional Parameters in Operations

An operation may [declare as optional](#) its return value and any of its parameters. The ordering rules for parameters remain unchanged: all input parameters must precede the output parameters.

The following Slice operation declares a mix of optional and required parameters, as well as an optional return value:

### Slice

```
interface Database
{
    optional(3) Record
    lookup(string key, optional(1) Filter f, out optional(2) Cursor matches);
}
```

We can invoke `lookup` in C++ as shown below:

## C++11

```
std::shared_ptr<DatabasePrx> proxy = ...;
Ice::optional<Cursor> matches;
auto rec = proxy->lookup("someKey", Ice::nullopt, matches);
if(rec)
{
    // lookup returned a value

    if(matches)
    {
        // there are more matches
    }
}
```

[Back to Top ^](#)

## Optional Data Members in Classes and Exceptions

[Data members](#) can be declared as optional in classes and exceptions. Structures do not support optional data members.

Consider the following example:

### Slice

```
class Account
{
    string accountNo;
    optional(9) Account referrer;
}
```

We can use Account in C# as shown below:

### C#

```
void printAccount(Account acct)
{
    Console.WriteLine("Account #: " + acct.accountNo);
    if(acct.referrer.HasValue)
    {
        Console.WriteLine("Referrer: " + acct.referrer.Value.accountNo);
    }
    else
    {
        Console.WriteLine("Referrer: None");
    }
}

Account a1 = new Account("100-21807", Ice.Util.None);
printAccount(a1);
Account a2 = new Account("165-75122", a1);
printAccount(a2);
```

Implicit conversion operators in C# allow us to pass a value of the declared type, or `Ice.Util.None` for an unset value, where an optional value is expected.

[Back to Top ^](#)

## Guidelines for using Optional Values

Keep the following guidelines in mind while using optional values:

- Consider an optional value as a logical unit composed of its tag and type. The parameter or member name is not included in the Ice encoding, therefore changes to the name do not affect on-the-wire compatibility. Changing the tag of a value is equivalent to adding a new value: the previous iteration of the value still exists as long as there is at least one deployed program that might use it.
- To change the type of an optional value without affecting compatibility with existing deployments, you must also change its tag. You are essentially deprecating the old value and adding a new one.
- Whenever you change the tag of a value, we recommend making a note of the old definition (such as in a Slice comment) so that you do not accidentally reuse its tag.
- Adding a new optional parameter or member does not affect compatibility with existing deployments as long as its tag is not currently in use.
- Removing an optional parameter or member does not affect compatibility with existing deployments if programs are written correctly to test for the presence of the value prior to using it.
- The order of declaration is important for required members and parameters, but is not important for optional members and parameters. If you wish to maintain compatibility with existing deployments, do not change the order of declaration for required members and parameters. Compatibility is not affected by changes to the order of optional members and parameters.

[Back to Top ^](#)

## Backward Compatibility with Encoding Version 1.0

Optional values require version 1.1 of the Ice encoding. If the intended receiver only supports version 1.0 of the encoding, the Ice run time in the sender will omit all optional parameters when marshaling the operation's parameters or results, and omit all optional data members. This behavior makes it possible for you to add optional parameters to an operation, or optional members to a class or exception, without affecting backward compatibility with existing deployments that use version 1.0 of the encoding, while gaining the ability to exchange optional values with new deployments that use version 1.1 of the encoding.

For example, suppose existing deployments use the following class definition:

### Slice

```
// Version 1
class Person
{
    string firstName;
    string lastName;
}
```

We can safely add an optional member without affecting compatibility:

### Slice

```
// Version 2
class Person
{
    string firstName;
    string lastName;
    optional(1) Date birthDate;
}
```

It is even safe to insert an optional member between two existing members:

### Slice

```
// Version 3
class Person
{
    string firstName;
    optional(2) string middleName;
    string lastName;
    optional(1) Date birthDate;
}
```

When sending an instance of `Person` to a receiver that supports only encoding version 1.0, the optional members are omitted and the encoded form matches our initial version of the class.

A similar situation exists for operations. For example, the following three operations are compatible when using encoding version 1.0:

## Slice

```
Person createPerson(string firstName, string lastName); // V1

Person createPerson(string firstName, string lastName, optional(1) Date birthDate); // V2

Person createPerson(string firstName, optional(2) string middleName, string lastName,
                    optional(1) Date birthDate); // V3
```

Note however that other types of changes can easily [break compatibility](#).

[Back to Top ^](#)

## Using Optional Values Efficiently

The Ice [encoding](#) uses a self-describing format to minimize the overhead associated with optional data members and allow for efficient decoding in a receiver. The amount of overhead required for a member depends on the member's type and its tag. The type contributes between one and five bytes of overhead, while the tag value contributes an additional zero to five bytes. Tag values less than 30 add no additional overhead; tag values between 30 and 254 add a single byte, and tag values of 255 or greater add five bytes.



The overhead associated with the member's type is out of your control, but you have direct control of the overhead for tags. To minimize this overhead, use tags in the range zero to 29.

The table below describes the overhead associated with several common Slice types:

Type	Overhead (bytes)
Primitive (including <code>string</code> )	1
Proxy	5
Object	1
Sequence of <code>byte</code> , <code>bool</code>	1
Sequence	2-5
Enumerator	1
Structure	2-5
Dictionary	2-5

Refer to the encoding specification for more details on marshaling optional data members.

[Back to Top ^](#)

### See Also

- [Data Encoding](#)
- [Optional Data Members](#)
- [Optional Parameters and Return Values](#)



Previous



Next