Java Mapping for Optional Data Members

On this page:

- Overview of Java Mapping for Optional Data Members
- Java Helper Classes for Optional Values

Overview of Java Mapping for Optional Data Members

The Java mapping for optional data members in Slice classes and exceptions uses a JavaBean-style API that provides methods to get, set, and clear a member's value, and test whether a value is set. Consider the following Slice definition:

Slice

```
class C
{
    string name;
    optional(2) string alternateName;
    optional(5) bool active;
};
```

The generated Java code provides the following API:

Java

```
public class C ...
{
    public C();
    public C(String name);
    public C(String name, String alternateName, boolean active);

    public String name;

    public String getAlternateName();
    public void setAlternateName(String v);
    public boolean hasAlternateName();
    public void clearAlternateName();
    public void optionalAlternateName(Ice.Optional<String> v);
    public Ice.Optional<String> optionalAlternateName();

    public boolean getActive();
    public void setActive(boolean v);
    public boolean isActive();
    public boolean hasActive();
    public void clearActive();
    public void optionalActive(Ice.BooleanOptional v);
    public Ice.BooleanOptional optionalActive();

    ...
}
```

If a class or exception declares any required data members, the generated class includes an overloaded constructor that accepts values for just the required members; optional members remain unset unless their Slice definitions specify a default value. Another overloaded constructor accepts values for all data members.

The `has` method allows you to test whether a member's value has been set, and the `clear` method removes any existing value for a member.

Calling a `get` method when the member's value has not been set raises `java.lang.IllegalStateException`.

The `optional` methods provide an alternate API that uses an `Optional` object to encapsulate the value, as discussed below.

Java Helper Classes for Optional Values

Ice defines the following classes to encapsulate optional values of primitive types:

- `Ice.BooleanOptional`
- `Ice.ByteOptional`
- `Ice.DoubleOptional`
- `Ice.FloatOptional`
- `Ice.IntOptional`
- `Ice.LongOptional`
- `Ice.ShortOptional`

These classes all share the same API for getting, setting, and testing an optional value. We'll use the `IntOptional` class as an example:

Java

```
public class IntOptional
{
    public IntOptional();                // Value is unset
    public IntOptional(int v);           // Value is set
    public IntOptional(IntOptional opt); // Copies the state of the argument
    public int get();                    // Raises IllegalStateException if unset
    public void set(int v);              // Value is set
    public void set(IntOptional opt);    // Copies the state of the argument
    public boolean isSet();
    public void clear();

    ...
}
```

The `Ice.Optional` generic class encapsulates values of reference types and offers an identical API:

Java

```
public class Optional<T>
{
    public Optional();                // Value is unset
    public Optional(T v);             // Value is set
    public Optional(Optional<T> opt); // Copies the state of the argument
    public T get();                   // Raises IllegalStateException if unset
    public void set(T v);             // Value is set
    public void set(Optional<T> opt); // Copies the state of the argument
    public boolean isSet();
    public void clear();


    public static <T> Optional<T> O(T v);
    public static BooleanOptional O(boolean v);
    public static ByteOptional O(byte v);
    public static ShortOptional O(short v);
    public static IntOptional O(int v);
    public static LongOptional O(long v);
    public static FloatOptional O(float v);
    public static DoubleOptional O(double v);

    ...
}
```

To improve the readability of code that creates optional values, the `Optional` class defines overloaded `O` methods that simplify the creation of these objects:

Java

```
import static Ice.Optional.O;

C obj = ...;
Ice.Optional<C> opt = O(obj);

Ice.IntOptional i = O(5);
```

See Also

- Optional Data Members