


# Parameter Passing in Java



Parameter passing on the server side follows the rules for the [client side](#). Additionally, every operation receives a trailing parameter of type `Current`. For example, the `name` operation of the `Node` interface has no parameters, but the corresponding `name` method of the servant interface has a single parameter of type `Current`. We will ignore this parameter for now.

 The parameter-passing rules change somewhat when using the [asynchronous mapping](#).

On this page:

- [Server-Side Mapping for Parameters in Java](#)
- [Thread-Safe Marshaling in Java](#)
  - [Solution 1: Copying](#)
  - [Solution 2: Copy on Write](#)
  - [Solution 3: Marshal Immediately](#)

## Server-Side Mapping for Parameters in Java

The servant mapping for operations is consistent with the proxy mapping. To illustrate the rules for the Java mapping, consider the following interface:

### Slice

```
module M
{
  interface Example
  {
    string op1();
    void op2(out string sout);
    string op3(string sin, out string sout);

    optional(1) string op4();
    void op5(out optional(1) string sout);
    optional(1) string op6(out optional(2) string sout);
  }
}
```

The generated skeleton interface looks like this:

## Java

```
public interface Example extends com.zeroc.Ice.Object
{
    public static class Op3Result
    {
        public Op3Result();
        public Op3Result(String returnValue, String sout);

        public String returnValue;
        public String sout;
    }

    public static class Op6Result
    {
        public Op6Result();
        public Op6Result(java.util.Optional<java.lang.String> returnValue,
            java.util.Optional<java.lang.String> sout);
        public Op6Result(java.lang.String returnValue, java.lang.String sout);

        public java.util.Optional<java.lang.String> returnValue;
        public java.util.Optional<java.lang.String> sout;
    }

    String op1(com.zeroc.Ice.Current current);
    String op2(com.zeroc.Ice.Current current);
    Example.Op3Result op3(String sin, com.zeroc.Ice.Current current);
    java.util.Optional<java.lang.String> op4(com.zeroc.Ice.Current current);
    java.util.Optional<java.lang.String> op5(com.zeroc.Ice.Current current);
    Example.Op6Result op6(com.zeroc.Ice.Current current);
}
```

You'll notice that `op1` and `op2` have same signature because the mapping rules state that an operation returning a single value shall use that type as its return value, regardless of whether the Slice operation declared it as the return type or as an out parameter. (The same is true for `op4` and `op5`.) The proxy and servant methods also share the `Result` types that are generated when an operation returns more than one value, as shown above for `op3` and `op6`. When at least one of the return value and out parameters is optional, the `Result` type provides two overload constructors that takes the return value followed by all out parameters: one with the `java.util.Optional` types, and one without `java.util.Optional` types; with the latter, a null value for an optional parameter is interpreted as meaning "not set".

The only difference between the client and server sides is the type of the extra trailing parameter.

[Back to Top ^](#)

## Thread-Safe Marshaling in Java

The marshaling semantics of the Ice run time present a subtle thread safety issue that arises when an operation returns data by reference. For Java applications, this can affect servant methods that return instances of Slice classes, structures, sequences, or dictionaries.

The potential for corruption occurs whenever a servant returns data by reference, yet continues to hold a reference to that data. For example, consider the following servant implementation:

## Java

```
public class GridI implements Grid
{
    GridI()
    {
        _grid = // ...
    }

    public int[][] getGrid(Current current)
    {
        return _grid;
    }

    public void setValue(int x, int y, int val, Current current)
    {
        _grid[x][y] = val;
    }

    private int[][] _grid;
}
```

Suppose that a client invoked the `getGrid` operation. While the Ice run time marshals the returned array in preparation to send a reply message, it is possible for another thread to dispatch the `setValue` operation on the same servant. This race condition can result in several unexpected outcomes, including a failure during marshaling or inconsistent data in the reply to `getGrid`. Synchronizing the `getGrid` and `setValue` operations would not fix the race condition because the Ice run time performs its marshaling outside of this synchronization.

## Solution 1: Copying

One solution is to implement accessor operations, such as `getGrid`, so that they return copies of any data that might change. There are several drawbacks to this approach:

- Excessive copying can have an adverse affect on performance.
- The operations must return deep copies in order to avoid similar problems with nested values.
- The code to create deep copies is tedious and error-prone to write.

## Solution 2: Copy on Write

Another solution is to make copies of the affected data only when it is modified. In the revised code shown below, `setValue` replaces `_grid` with a copy that contains the new element, leaving the previous contents of `_grid` unchanged:

## Java

```
public class GridI implements Grid
{
    ...

    public synchronized int[][] getGrid(Current current)
    {
        return _grid;
    }

    public synchronized void setValue(int x, int y, int val, Current current)
    {
        int[][] newGrid = // shallow copy...
        newGrid[x][y] = val;
        _grid = newGrid;
    }

    ...
}
```

This allows the Ice run time to safely marshal the return value of `getGrid` because the array is never modified again. For applications where data is read more often than it is written, this solution is more efficient than the previous one because accessor operations do not need to make copies. Furthermore, intelligent use of shallow copying can minimize the overhead in mutating operations.

## Solution 3: Marshal Immediately

Finally, a third approach is to modify the servant mapping using metadata in order to force the marshaling to occur immediately within your synchronization. Annotating a Slice operation with the `marshaled-result` metadata directive changes the signature of the corresponding servant method, but only if that operation returns one or more of the mutable types listed earlier. The metadata directive has the following effects:

- For an operation `op` that returns multiple values and at least one of those values has a mutable type, the name of the generated `OpResult` class becomes `OpMarshaledResult` instead and the return type of the servant method becomes `OpMarshaledResult`.
- For an operation `op` that returns a single value whose type is mutable, the Slice compiler generates an `OpMarshaledResult` class and the return type of the servant method becomes `OpMarshaledResult`.
- The constructor for `OpMarshaledResult` takes an extra argument of type `Current`. The servant must supply the `Current` in order for the results to be marshaled correctly.

The metadata directive also affects the [asynchronous mapping](#) but has no effect on the proxy mapping, nor does it affect the servant mapping of Slice operations that return `void` or return only immutable values.



You can also annotate an interface with the `marshaled-result` metadata and it will be applied to all of the interface's operations.

After applying the metadata, we can now implement the `Grid` servant as follows:

### Java

```
public class GridI implements Grid
{
    ...

    public synchronized Grid.GetGridMarshaledResult getGrid(Current current)
    {
        return new Grid.GetGridMarshaledResult(_grid, curr); // _grid is marshaled immediately
    }

    public synchronized void setValue(int x, int y, int val, Current current)
    {
        _grid[x][y] = val; // this is safe
    }

    ...
}
```

Here are more examples to demonstrate the mapping:

## Slice

```
class C { ... }
struct S { ... }
sequence<string> Seq;

interface Example
{
    C getC();

    ["marshaled-result"]
    C getC2();

    void getS(out S val);

    ["marshaled-result"]
    void getS2(out S val);

    string getValues(string name, out Seq val);

    ["marshaled-result"]
    string getValues2(string name, out Seq val);

    ["amd", "marshaled-result"]
    string getValuesAMD(string name, out Seq val);
}
```

Review the generated code below to see the changes that the presence of the metadata causes in the servant method signatures:

## Java

```
public interface Example extends com.zeroc.Ice.Object
{
    public static class GetC2MarshaledResult
    {
        public GetC2MarshaledResult(C returnValue, Current current);
        ...
    }

    public static class GetS2MarshaledResult
    {
        public GetS2MarshaledResult(S returnValue, Current current);
        ...
    }

    public static class GetValuesResult
    {
        public GetValuesResult();
        public GetValuesResult(String returnValue, String[] val);

        public String returnValue;
        public String[] val;
    }

    public static class GetValues2MarshaledResult
    {
        public GetValues2MarshaledResult(String returnValue, String[] val, Current current);
        ...
    }

    public static class GetValuesAMDResult
    {
        public GetValuesAMDResult()
        {
        }

        public GetValuesAMDResult(String returnValue, String[] val)
        {
            this.returnValue = returnValue;
            this.val = val;
        }

        public String returnValue;
        public String[] val;
    }

    public static class GetValuesAMDMarshaledResult implements com.zeroc.Ice.MarshaledResult
    {
        public GetValuesAMDMarshaledResult(String returnValue, String[] val, com.zeroc.Ice.Current current);
        ...
    }

    C getC(com.zeroc.Ice.Current current);
    GetC2MarshaledResult getC2(com.zeroc.Ice.Current current);
    S getS(com.zeroc.Ice.Current current);
    GetS2MarshaledResult getS2(com.zeroc.Ice.Current current);
    GetValuesResult getValues(String name, com.zeroc.Ice.Current current);
    GetValues2MarshaledResult getValues2(String name, com.zeroc.Ice.Current current);
    java.util.concurrent.CompletionStage<GetValuesAMDMarshaledResult> getValuesAMDAsync(String name, com.zeroc.
Ice.Current current);
}
```

[Back to Top ^](#)

## See Also

- [Server-Side Java Mapping for Interfaces](#)

- [Java Mapping for Operations](#)
- [Raising Exceptions in Java](#)
- [The Current Object](#)

