

Class Inheritance Semantics

Classes use the same pass-by-value semantics as [structures](#). If you pass a class instance to an operation, the class and all its members are passed. The usual type compatibility rules apply: you can pass a derived instance where a base instance is expected. If the receiver has static type knowledge of the actual derived run-time type, it receives the derived instance; otherwise, if the receiver does not have static type knowledge of the derived type, the instance is sliced to the base type. For an example, suppose we have the following definitions:

```

Slice

// In file Clock.ice:

class TimeOfDay {
    short hour;           // 0 - 23
    short minute;        // 0 - 59
    short second;        // 0 - 59
};

interface Clock {
    TimeOfDay getTime();
    void setTime(TimeOfDay time);
};

// In file DateTime.ice:

#include <Clock.ice>

class DateTime extends TimeOfDay {
    short day;           // 1 - 31
    short month;         // 1 - 12
    short year;          // 1753 onwards
};

```

Because `DateTime` is a sub-class of `TimeOfDay`, the server can return a `DateTime` instance from `getTime`, and the client can pass a `DateTime` instance to `setTime`. In this case, if both client and server are linked to include the code generated for both `Clock.ice` and `DateTime.ice`, they each receive the actual derived `DateTime` instance, that is, the actual run-time type of the instance is preserved.

Contrast this with the case where the server is linked to include the code generated for both `Clock.ice` and `DateTime.ice`, but the client is linked only with the code generated for `Clock.ice`. In other words, the server understands the type `DateTime` and can return a `DateTime` instance from `getTime`, but the client only understands `TimeOfDay`. In this case, the derived `DateTime` instance returned by the server is sliced to its `TimeOfDay` base type in the client. (The information in the derived part of the instance is simply lost to the client.)

Class hierarchies are useful if you need polymorphic *values* (instead of polymorphic *interfaces*). For example:

```

Slice

class Shape {
    // Definitions for shapes, such as size, center, etc.
};

class Circle extends Shape {
    // Definitions for circles, such as radius...
};

class Rectangle extends Shape {
    // Definitions for rectangles, such as width and length...
};

sequence<Shape> ShapeSeq;

interface ShapeProcessor {
    void processShapes(ShapeSeq ss);
};

```

Note the definition of `ShapeSeq` and its use as a parameter to the `processShapes` operation: the class hierarchy allows us to pass a polymorphic sequence of shapes (instead of having to define a separate operation for each type of shape).

The receiver of a `ShapeSeq` can iterate over the elements of the sequence and down-cast each element to its actual run-time type. (The receiver can also ask each element for its `type ID` to determine its type.)

See Also

- [Structures](#)
- [Type IDs](#)