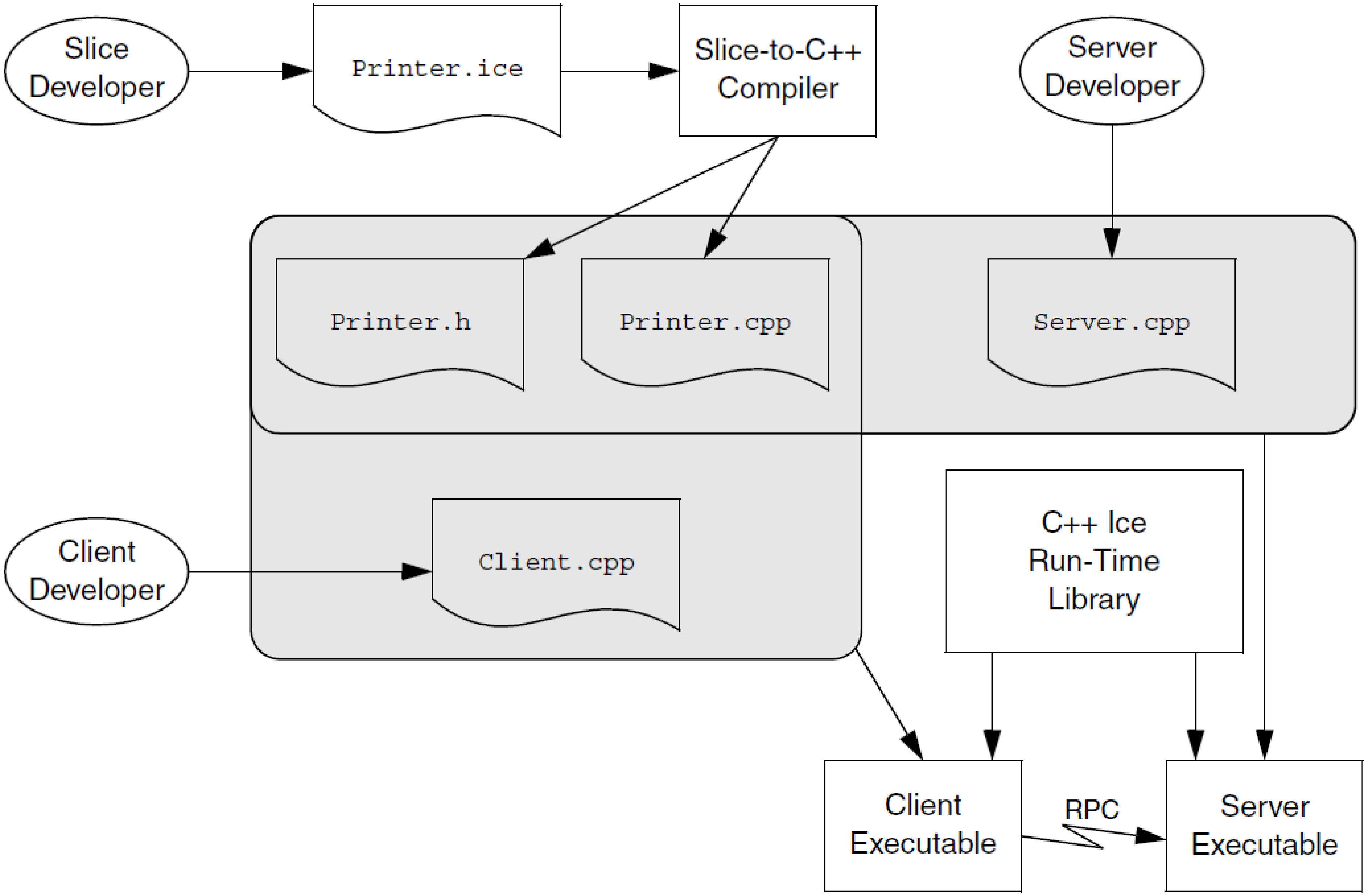
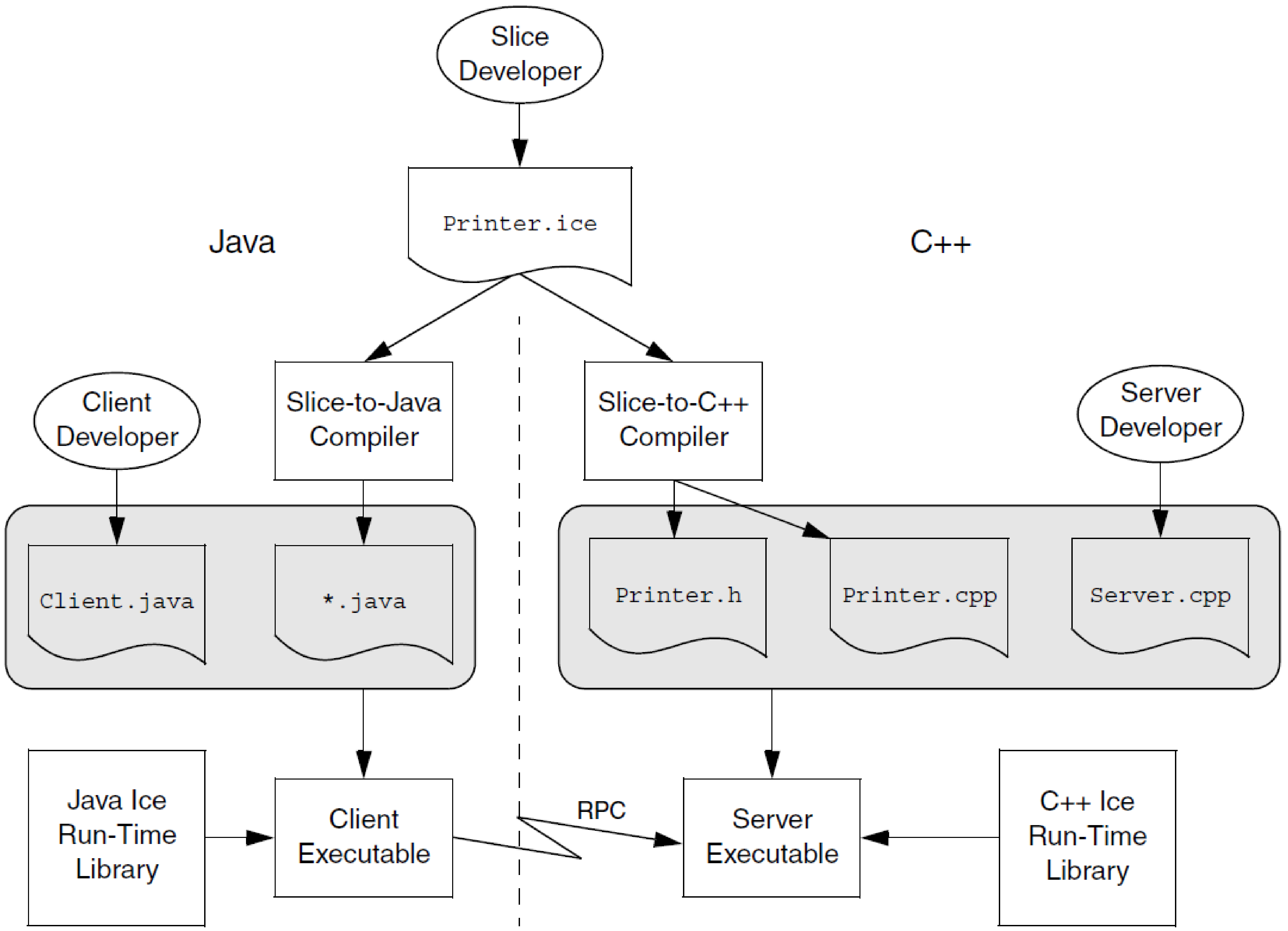
The Slice Language

Single Development Environment for Client and Server



Different Development Environments for Client and Server



Slice Source FIles

- Files containing Slice definitions must end in a .ice file extension, for example, `Clock.ice` is a valid file name. Other file extensions are rejected by the compilers.
- Slice is a free-form language so you can use spaces, horizontal and vertical tab stops, form feeds, and newline characters to lay out your code in any way you wish.
- Slice files use the (7-bit) ASCII character set.
- Slice is preprocessed by the C++ preprocessor, so you can use the usual preprocessor directives, such as #include and macro definitions.
  - Slice permits #include directives only at the beginning of a file, before any Slice definitions.
  - If you include a path separator in a #include directive, you must use a forward slash.
- Slice constructs, such as modules, interfaces, or type definitions, can appear in any order you prefer.
  - Identifiers must be declared before they can be used.

Lexical Rules

- Slice definitions permit both the C and the C++ style of writing comments:

```
/*
 * C-style comment.
 */
// C++-style comment extending to the end of this line.
```

- Slice uses a number of keywords, which must be spelled in lowercase.
  - Examples: **`class`** and **`dictionary`**
  - Two exceptions: **`Object`** and **`LocalObject`**
- Identifiers begin with an alphabetic character followed by any number of alphabetic characters or digits
  - Slice identifiers cannot contain underscores.
- Identifiers are case-insensitive but must be capitalized consistently.
- Slice reserves the identifier Ice and all identifiers beginning with Ice (in any capitalization) for the Ice implementation.

Modules

- Prevent pollution of the global namespace

```
module ZeroC {
    module Client {
        // Definitions here...
    };
    module Server {
        // Definitions here...
    };
};
```

- A module can contain any legal Slice construct, including other module definitions.
- Slice requires all definitions to be nested inside a module, that is, you cannot define anything other than a module at global scope. For example, the following is illegal:

```
interface I {   // Error: only modules can appear at global scope
    // ...
};
```

Modules

- Modules can be reopened:

```
module ZeroC {
    // Definitions here...
};

// Possibly in a different source file:

module ZeroC {  // OK, reopened module
    // More definitions here...
};
```

- APIs for the Ice run time, apart from a small number of language-specific calls that cannot be expressed in Ice, are defined in the Ice module.

Basic Slice Types

Type	Range of Mapped Type	Size of Mapped Type
`bool`	`false` or `true`	$\geq$ 1bit
`byte`	–128–127[a]	$\geq$ 8 bits
`short`	-2^{15} to $2^{15}-1$	$\geq$ 16 bits
`int`	-2^{31} to $2^{31}-1$	$\geq$ 32 bits
`long`	-2^{63} to $2^{63}-1$	$\geq$ 64 bits
`float`	IEEE single-precision	$\geq$ 32 bits
`double`	IEEE double-precision	$\geq$ 64 bits
`string`	All Unicode characters, excluding the character with all bits zero.	Variable-length

Enumerations

- A Slice enumerated type definition looks like the C++ version:
  - `enum Fruit { Apple, Pear, Orange };`
- Slice does not define how ordinal values are assigned to enumerators.
- Slice does not permit you to control the ordinal values of enumerators, the following Slice code is illegal:
  - `enum Fruit { Apple = 0, Pear = 7, Orange = 2 };`
- Slice enumerators enter the enclosing namespace, so the following is illegal:
  - `enum Fruit { Apple, Pear, Orange };`
  - `enum ComputerBrands { Apple, IBM, Sun, HP }; // Apple redefined`
- Slice does not permit empty enumerations.

Structures

- Slice supports structures containing one or more named members of arbitrary type, including user-defined complex types.

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

- Structure definitions form a namespace, so the names of the structure members need to be unique only within their enclosing structure.

Structures

- Data member definitions using a named type are the only construct that can appear inside a structure. It is impossible to, for example, define a structure inside a structure:

```
struct TwoPoints {
  struct Point { // Illegal!
    short x;
    short y;
  };
  Point coord1;
  Point coord2;
};
```

- This rule applies to Slice in general: type definitions cannot be nested, except for modules

Structures

- The structure from the previous slide can be defined as follows:

```
struct Point {
  short x;
  short y;
};

struct TwoPoints { // Legal (and cleaner!)
  Point coord1;
  Point coord2;
};
```

Sequences

- Sequences are variable-length collections of elements:
  - **`sequence<Fruit> FruitPlatter;`**
- Sequences can contain elements that are themselves sequences. This arrangement allows you to create lists of lists:
  - **`sequence<FruitPlatter> FruitBanquet;`**
- Sequences can be used to model the optional field:

```
sequence<long> SerialOpt;
struct Part {
  string name;
  string description;
  // ...
  SerialOpt serialNumber; // optional: zero or one element
};
```

Dictionaries

- A dictionary is a mapping from a key type to a value type:

```
struct Employee {
  long number;
  string firstName;
  string lastName;
};
dictionary<long, Employee> EmployeeMap;
```

- The key type of a dictionary need not be an integral type.
  - **`dictionary<string, string> WeekdaysEnglishToGerman;`**
- The value type of a dictionary can be any user-defined type. The key type of a dictionary is limited to one of the following types:
  - Integral types (**`byte`**, **`short`**, **`int`**, **`long`**, **`bool`**, and enumerated types)
  - **`string`**
  - Structures containing only data members of integral type or **`string`**

Constant Definitions and Literals

- Constant definitions must be of one of the following types:
  - An integral type (**`bool`**, **`byte`**, **`short`**, **`int`**, **`long`**, or an enumerated type)
  - **`float`** or **`double`**
  - **`string`**
- Examples:
  - **`const bool AppendByDefault = true;`**
  - **`const byte LowerNibble = 0x0f;`**
  - **`const string Advice = "Don't Panic!";`**
  - **`const short TheAnswer = 42;`**
  - **`const double PI = 3.1416;`**
  - **`enum Fruit { Apple, Pear, Orange };`**
  - **`const Fruit FavoriteFruit = Pear;`**

Constant Definitions and Literals

- The syntax for literals is the same as for C++ and Java (with a few minor exceptions):
  - Boolean constants can only be initialized with the keywords **`false`** and **`true`**. (You cannot use 0 and 1 to represent false and true.)
  - As for C++, integer literals can be specified in decimal, octal, or hexadecimal notation. For example:
    - **`const byte TheAnswer = 42;`**
    - **`const byte TheAnswerInOctal = 052;`**
    - **`const byte TheAnswerInHex = 0x2A; // or 0x2a`**
  - If you interpret byte as a number instead of a bit pattern, you may get different results in different languages. For example, for C++, **`byte`** maps to **`unsigned char`** whereas, for Java, **`byte`** maps to **`byte`**, which is a signed type.

Constant Definitions and Literals

- The suffixes to indicate long and unsigned constants (l, L, u, U, used by C++) are illegal:
  - `const long Wrong = 0u; // Syntax error`
  - `const long WrongToo = 1000000L; // Syntax error`
- Floating-point literals use C++ syntax, except that you cannot use an l or L suffix to indicate an extended floating-point constant; however, f and F are legal (but are ignored).
  - `const float P1 = -3.14f; // Integer & fraction, with suffix`
  - `const float P2 = +3.1e-3; // Integer, fraction, and exponent`
  - `const float P3 = .1; // Fraction part only`
  - `const float P4 = 1.; // Integer part only`
  - `const float P5 = .9E5; // Fraction part and exponent`
  - `const float P6 = 5e2; // Integer part and exponent`

Constant Definitions and Literals

- String literals support the same escape sequences as C++.
  - `const string AnOrdinaryString = "Hello World!";`
  - `const string DoubleQuote = "\"";`
  - `const string TwoSingleQuotes = "'\'"; // ' and \' are OK`
  - `const string Newline = "\n";`
  - `const string CarriageReturn = "\r";`
  - `const string HorizontalTab = "\t";`
  - `const string VerticalTab = "\v";`
  - `const string FormFeed = "\f";`
  - `const string Alert = "\a";`
  - `const string Backspace = "\b";`
  - `const string QuestionMark = "\?";`
  - `const string Backslash = "\\";`
  - `const string OctalEscape = "\007"; // Same as \a`
  - `const string HexEscape = "\x07"; // Ditto`
  - `const string UniversalCharName = "\u03A9"; // Greek Omega`

Constant Definitions and Literals

- As for C++, adjacent string literals are concatenated:

```
const string MSG1 = "Hello World!";
const string MSG2 = "Hello" " " "World!"; // Same message
/*
* Escape sequences are processed before concatenation,
* so the string below contains two characters,
* '\xa' and 'c'.
*/
const string S = "\xa" "c";
```

- Note that Slice has no concept of a null string:
  - `const string nullString = 0; // Illegal!`
- Null strings simply do not exist in Slice and, therefore, do not exist as a legal value for a string anywhere in the Ice platform.

Interfaces, Operations, and Exceptions

- The central focus of Slice is on defining interfaces, for example:

```
struct TimeOfDay {
  short hour; // 0 - 23
  short minute; // 0 - 59
  short second; // 0 - 59
};
interface Clock {
  TimeOfDay getTime();
  void setTime(TimeOfDay time);
};
```

Parameters and Return Values

- An operation definition must contain a return type and zero or more parameter definitions.
  - You must use void to indicate that an operation returns no value - there is no default return type for Slice operations.
- An operation can have one or more input parameters.

```
interface CircadianRhythm {
  void setSleepPeriod(TimeOfDay startTime,
                      TimeOfDay stopTime);
  // ...
};
```

- The parameter name is mandatory. The following is in error:

```
interface CircadianRhythm {
  void setSleepPeriod(TimeOfDay, TimeOfDay);
                                          // Error!
  // ...
};
```

Parameters and Return Values

- By default, parameters are sent from the client to the server, that is, they are input parameters. To pass a value from the server to the client, you can use an output parameter, indicated by the out keyword.
- An alternative way to define the getTime operation
  - `void getTime(out TimeOfDay time);`
- You can use multiple output parameters:

```
interface CircadianRhythm {
  void setSleepPeriod(TimeOfDay startTime,
                      TimeOfDay stopTime);
  void getSleepPeriod(out TimeOfDay startTime,
  out TimeOfDay stopTime);
  // ...
};
```

Parameters and Return Values

- If you have both input and output parameters for an operation, the output parameters must follow the input parameters:

```
void changeSleepPeriod( TimeOfDay startTime, // OK
                        TimeOfDay stopTime,
                        out TimeOfDay prevStartTime,
                        out TimeOfDay prevStopTime);
void changeSleepPeriod(out TimeOfDay prevStartTime,
                       out TimeOfDay prevStopTime,
                       TimeOfDay startTime, // Error
                       TimeOfDay stopTime);
```

- Slice does not support parameters that are both input and output parameters (call by reference).
- For operations that return only a single value, it is common to return the value from the operation instead of using an out-parameter.
- For operations that return multiple values, it is common to return all values as out-parameters and to use a return type of void.
  - The rule is not all that clear-cut because operations with multiple output values can have one particular value that is considered more “important” than the remainder.

Overloading

- Slice does not support any form of overloading of operations.

```
interface CircadianRhythm {
  void modify(TimeOfDay startTime,
              TimeOfDay endTime);
  void modify( TimeOfDay startTime, // Error
               TimeOfDay endTime,
               out timeOfDay prevStartTime,
               out TimeOfDay prevEndTime);
};
```

- Operations in the same interface must have different names, regardless of what type and number of parameters they have.

Idempotent Operations

- Some operations do not modify the state of the object they operate on. They are the conceptual equivalent of C++ **`const`** member functions. Similary, **`setTime`** does modify the state of the object, but is idempotent. You can indicate this in Slice as follows:

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

- An operation is idempotent if two successive invocations of the operation have the same effect as a single invocation.
  - **`x = 1;`** is an idempotent operation because it does not matter whether it is executed once or twice
    - Either way, x ends up with the value 1.
  - **`x += 1;`** is not an idempotent operation
    - Executing it twice results in a different value for x than executing it once.
- Obviously, any read-only operation is idempotent.

User Exceptions

- Slice allows to define user exceptions to indicate error conditions to the client. For example:

```
exception Error {}; // Empty exceptions are legal
exception RangeError {
  TimeOfDay errorTime;
  TimeOfDay minTime;
  TimeOfDay maxTime;
};
```

  - A user exception is much like a structure in that it contains a number of data members. However, unlike structures, exceptions can have zero data members, that is, be empty.

User Exceptions

- Operations use an exception specification to indicate the exceptions that may be returned to the client:

```
interface Clock {
  idempotent TimeOfDay getTime();
  idempotent void setTime(TimeOfDay time)
                                          throws RangeError, Error;
};
```

- This definition indicates that the **`setTime`** operation may throw either a **`RangeError`** or an **`Error`** user exception (and no other type of exception).
  - If the client receives a **`RangeError`** exception, the exception contains the **`TimeOfDay`** value that was passed to **`setTime`** and caused the error, as well as the minimum and maximum time values that can be used.
  - If **`setTime`** failed because of an error not caused by an illegal parameter value, it throws **`Error`**.
    - The client will have no idea what exactly it was that went wrong - it simply knows that the operation did not work.

User Exceptions

- An operation can throw only those user exceptions that are listed in its exception specification. If, at run time, the implementation of an operation throws an exception that is not listed in its exception specification, the client receives a runtime exception to indicate that the operation did something illegal. To indicate that an operation does not throw any user exception, simply omit the exception specification.
- Exceptions are not first-class data types and first-class data types are not exceptions:
  - You cannot pass an exception as a parameter value.
  - You cannot use an exception as the type of a data member.
  - You cannot use an exception as the element type of a sequence.
  - You cannot use an exception as the key or value type of a dictionary.
  - You cannot throw a value of non-exception type (such as a value of type **`int`** or **`string`**).

Exception Inheritance

- Exceptions support inheritance.

```
exception ErrorBase {
  string reason;
};
enum RTError {
  DivideByZero, NegativeRoot, IllegalNull /* ... */
};
exception RuntimeError extends ErrorBase {
  RTError err;
};
enum LError { ValueOutOfRange, ValuesInconsistent, /* ... */ };
exception LogicError extends ErrorBase {
  LError err;
};
exception RangeError extends LogicError {
  TimeOfDay errorTime;
  TimeOfDay minTime;
  TimeOfDay maxTime;
};
```

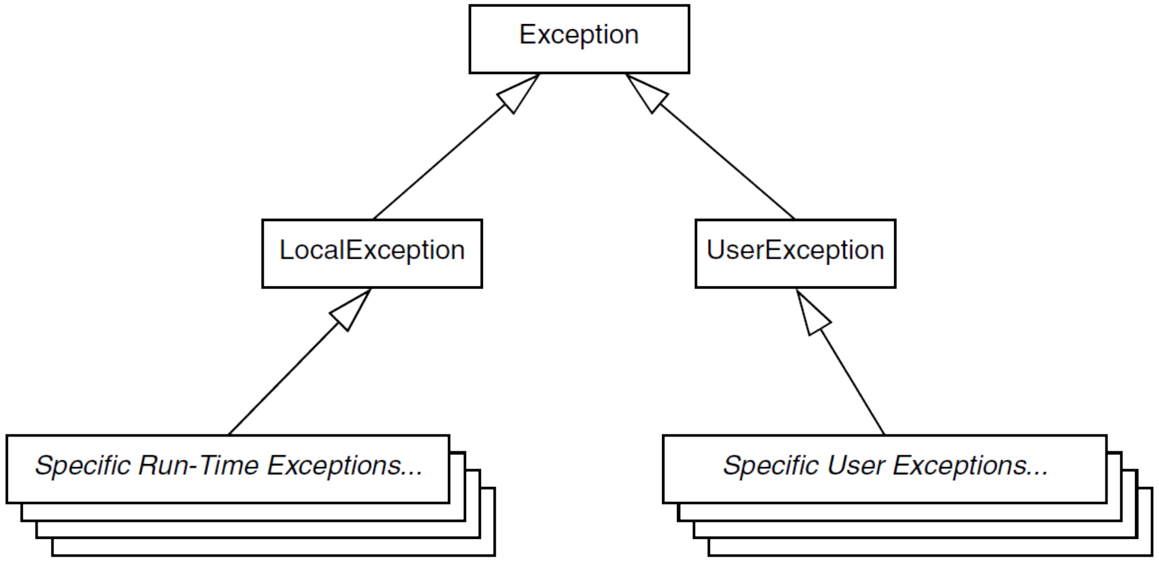
Exception Inheritance

- If the exception specification of an operation indicates a specific exception type, at run time, the implementation of the operation may also throw more derived exceptions.

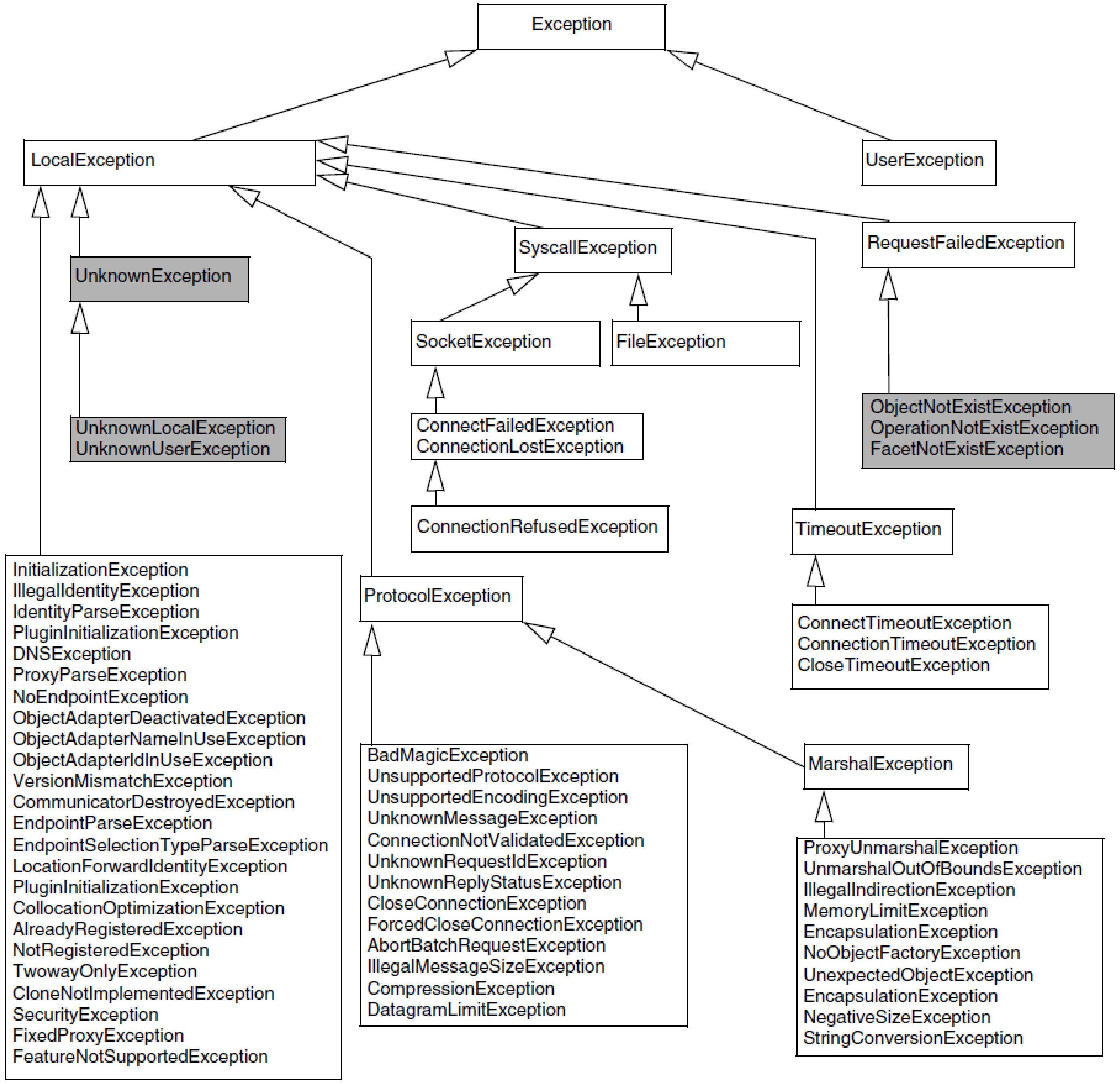
```
exception Base {
  // ...
};
exception Derived extends Base {
  // ...
};
interface Example {
  void op() throws Base; // May throw Base or Derived
};
```

Ice Run-Time Exceptions

- In addition to any user exceptions that are listed in an operation's exception specification, an operation can also throw Ice run-time exceptions.
- The exception specification of an operation must not list any run-time exceptions.
- All the Ice run-time and user exceptions are arranged in the following inheritance hierarchy:



The Complete Hierarchy of The Ice Run-Time Exceptions



Interface Semantics and Proxies

- Building on the Clock example, we can create definitions for a world-time server:

```
exception GenericError {
  string reason;
};
struct TimeOfDay {
  short hour; // 0 - 23
  short minute; // 0 - 59
  short second; // 0 - 59
};
exception BadTimeVal extends GenericError {};
interface Clock {
  idempotent TimeOfDay getTime();
  idempotent void setTime(TimeOfDay time) throws BadTimeVal;
};
dictionary<string, Clock*> TimeMap; // Time zone name to clock map
exception BadZoneName extends GenericError {};
interface WorldTime {
  idempotent void addZone(string zoneName, Clock* zoneClock);
  void removeZone(string zoneName) throws BadZoneName;
  idempotent Clock* findZone(string zoneName) throws BadZoneName;
  idempotent TimeMap listZones();
  idempotent void setZones(TimeMap zones);
};
```

The Proxy Operator

- **`addZone`** accepts a parameter of type **`Clock*`** and **`findZone`** returns a parameter of type **`Clock*`**.
  - Interfaces are types in their own right and can be passed as parameters. The * operator is known as the proxy operator.
    - Its left-hand argument must be an interface (or class) and its return type is a proxy.
    - A proxy is like a pointer that can denote an object.
    - The semantics of proxies are very much like those of C++ class instance pointers:
      - A proxy can be null.
      - A proxy can dangle (point at an object that is no longer there)
      - Operations dispatched via a proxy use late binding: if the actual run-time type of the object denoted by the proxy is more derived than the proxy's type, the implementation of the most-derived interface will be invoked.
- When a client passes a **`Clock`** proxy to the **`addZone`** operation, the proxy denotes an actual **`Clock`** object in a server.
- The **`Clock`** Ice object denoted by that proxy may be implemented in the same server process as the **`WorldTime`** interface, or in a different server process.

Interface Inheritance

- Interfaces support inheritance.
- We could extend our world-time server to support the concept of an alarm clock:

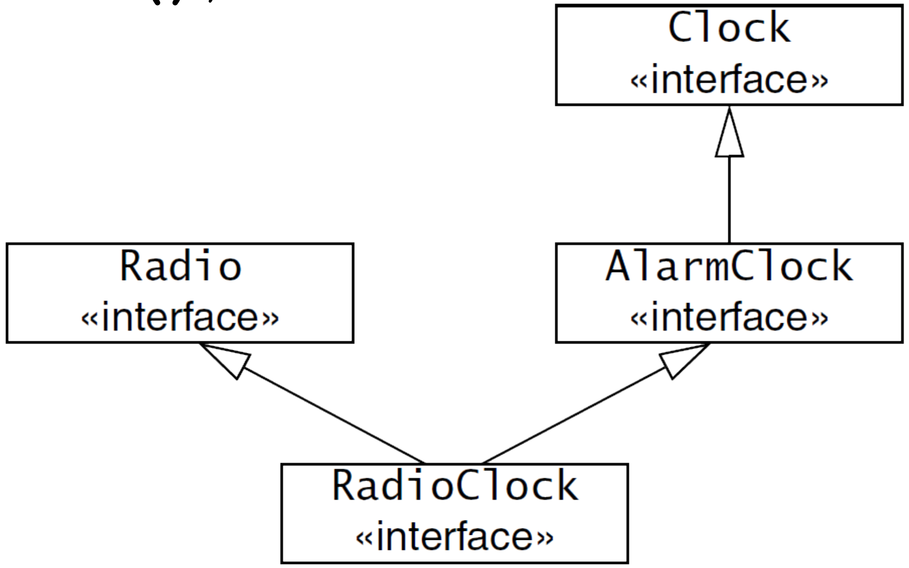
```
interface AlarmClock extends Clock {
  idempotent TimeOfDay getAlarmTime();
  idempotent void setAlarmTime(TimeOfDay alarmTime)
  throws BadTimeVal;
};
```

- **`AlarmClock`** is a subtype of **`Clock`** and an **`AlarmClock`** proxy can be substituted wherever a **`Clock`** proxy is expected.
- An **`AlarmClock`** supports the same **`getTime`** and **`setTime`** operations as a **`Clock`** but also supports the **`getAlarmTime`** and **`setAlarmTime`** operations.

Multiple Inheritance

- We can construct a radio alarm clock as follows:

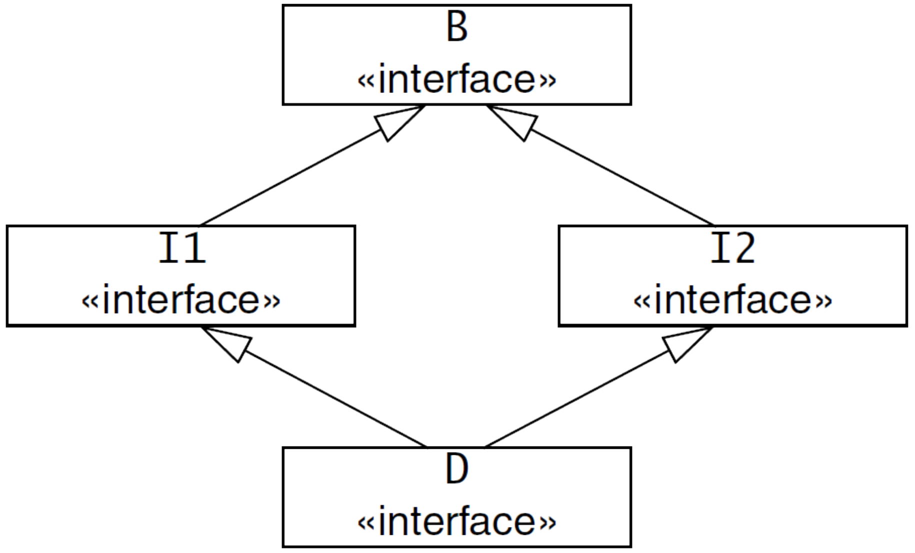
```
interface Radio {
  void setFrequency(long hertz) throws GenericError;
  void setVolume(long dB) throws GenericError;
};
enum AlarmMode { RadioAlarm, BeepAlarm };
interface RadioClock extends Radio, AlarmClock {
  void setMode(AlarmMode mode);
AlarmMode getMode();
};
```



Multiple Inheritance

- Interfaces that inherit from more than one base interface may share a common base interface.

```
interface B { /* ... */ };
interface I1 extends B { /* ... */ };
interface I2 extends B { /* ... */ };
interface D extends I1, I2 { /* ... */ };
```



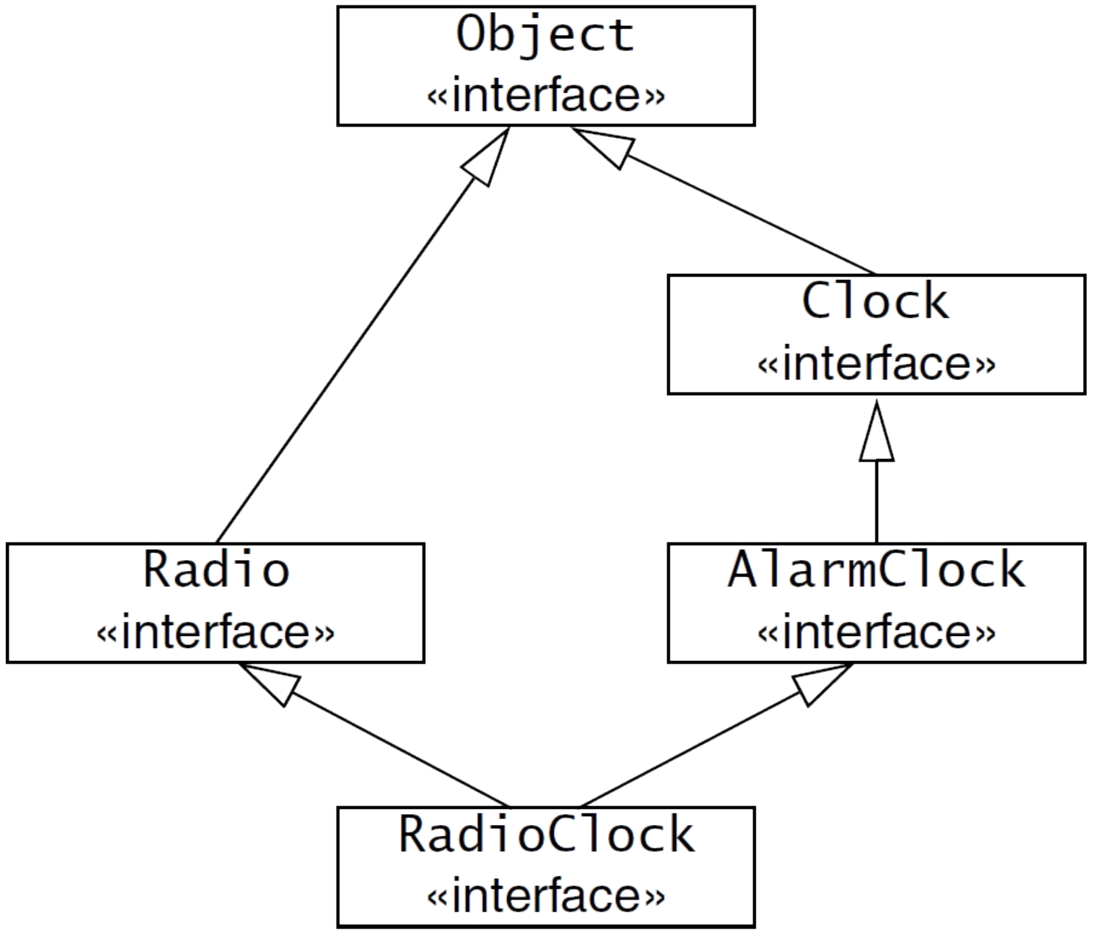
Interface Inheritance Limitations

- If an interface uses multiple inheritance, it must not inherit the same operation name from more than one base interface.
- The following definition is illegal:

```
interface Clock {
  void set(TimeOfDay time); // set time
};
interface Radio {
  void set(long hertz); // set frequency
};
interface RadioClock extends Radio, Clock { // Illegal!
  // ...
};
```

Implicit Inheritance from Object

- All Slice interfaces are ultimately derived from **`Object`**.



Interface Inhertance

- Because all interfaces have a common base interface, we can pass any type of interface as that type.

```
interface ProxyStore {
  idempotent void putProxy(string name, Object* o);
  idempotent Object* getProxy(string name);
};
```

- Inheritance from type Object is always implicit. For example, the following Slice definition is illegal:
  - `interface MyInterface extends Object { /* ... */ }; // Error!`

Self-Referential Interfaces

- Proxies have pointer semantics, so we can define self-referential interfaces.

```
interface Link {
    idempotent SomeType getValue();
    idempotent Link* next();
};
```

Empty Interfaces

- The following Slice definition is legal:
  - `interface Empty {};`
- The Slice compiler will compile this definition without complaint. An interesting question is: “why would I need an empty interface?” In most cases, empty interfaces are an indication of design errors.

```
interface ThingBase {};
interface Thing1 extends ThingBase {
// Operations here...
};
interface Thing2 extends ThingBase {
// Operations here...
};
```

Interface Versus Implementation Inheritance

- Slice interface inheritance applies only to interfaces.
  - In particular, if two interfaces are in an inheritance relationship, this in no way implies that the implementations of those interfaces must also inherit from each other.
  - You can choose to use implementation inheritance when you implement your interfaces, but you can also make the implementations independent of each other.
- Slice inheritance simply establishes type compatibility.
  - It says nothing about how interfaces are implemented and, therefore, keeps implementation choices open to whatever is most appropriate for your application.

Forward Declarations

- Both interfaces and classes can be forward declared.
- Forward declarations permit the creation of mutually dependent objects:

```
module Family {
  interface Child; // Forward declaration
  sequence<Child*> Children; // OK
  interface Parent {
    Children getChildren(); // OK
  };
  interface Child { // Definition
    Parent* getMother();
    Parent* getFather();
  };
};
```

Forward Declarations

- You cannot inherit from a forward-declared interface or class until after its definition has been seen by the compiler:

```
interface Base; // Forward declaration
interface Derived1 extends Base {}; // Error!
interface Base {}; // Definition
interface Derived2 extends Base {};
                                             // OK, definition was seen
```

Type IDs

- Each user-defined Slice type has an internal type identifier, known as its type ID.
  - The type ID is simply the fully-qualified name of each type. For example, the type ID of the **`Child`** interface in the preceding example is **`::Family::Children::Child`**.
  - All type IDs for user-defined types start with a leading `::`, so the type ID of the **`Family`** module is **`::Family`** (not **`Family`**).
  - In general, a type ID is formed by starting with the global scope (`::`) and forming the fully-qualified name of a type by appending each module name in which the type is nested, and ending with the name of the type itself; the components of the type ID are separated by `::`.
  - The type ID of a proxy is formed by appending a * to the type ID of an interface or class. For example, the type ID of a **`Child`** proxy is **`::Family::Children::Child*`**.

Type IDs

- The type ID of the Slice **`Object`** type is **`::Ice::Object`** and the type ID of an **`Object`** proxy is **`::Ice::Object*`**.
- The type IDs for the remaining built-in types, such as **`int`**, **`bool`**, and so on, are the same as the corresponding keyword.
  - The type ID of **`int`** is int,
  - The type ID of **`string`** is **`string`**.
- Type IDs are used internally by the Ice run time as a unique identifier for each type.
  - When an exception is raised, the marshaled form of the exception that is returned to the client is preceded by its Type ID on the wire.
  - The clientside run time first reads the Type ID and, based on that, unmarshals the remainder of the data as appropriate for the type of the exception.
- Type IDs are also used by the **`ice_isA`** operation

Operations on `Object`

- The **`Object`** interface has a number of operations. We cannot define type **`Object`** in Slice because **`Object`** is a keyword.
- The definition of Object would look like below if it were legal:

```
sequence<string> StrSeq;
interface Object { // "Pseudo" Slice!
  idempotent void ice_ping();
  idempotent bool ice_isA(string typeID);
  idempotent string ice_id();
  idempotent StrSeq ice_ids();
  // ...
};
```

`ice_ping`

- All interfaces support the **`ice_ping`** operation.
- That operation is useful for debugging
  - It provides a basic reachability test for an object:
    - If the object exists and a message can successfully be dispatched to the object, ice_ping simply returns without error.
    - If the object cannot be reached or does not exist, **`ice_ping`** throws a run-time exception that provides the reason for the failure.

ice_isA

- The **`ice_isA`** operation accepts a type identifier (such as the identifier returned by **`ice_id`**) and tests whether the target object supports the specified type, returning true if it does. You can use this operation to check whether a target object supports a particular type.
- Assume that you are holding a proxy to a target object of type **`AlarmClock`**. Table below illustrates the result of calling **`ice_isA`** on that proxy with various arguments.

Argument	Result
`::Ice::Object`	`true`
`::Times::Clock`	`true`
`::Times::AlarmClock`	`true`
`::Times::Radio`	`false`
`::Times::RadioClock`	`false`

Operations on `Object`

- **`ice_id`**
  - The **`ice_id`** operation returns the type ID of the most derived type of an interface.
- **`ice_ids`**
  - The **`ice_ids`** operation returns a sequence of type IDs that contains all of the type IDs supported by an interface. For example, for the **`RadioClock`** interface, **`ice_ids`** returns a sequence containing the type IDs
  - **`::Ice::Object`**, **`::Times::Clock`**, **`::Times::AlarmClock`**, **`::Times::Radio`**, and **`::Times::RadioClock`**.

Local Types

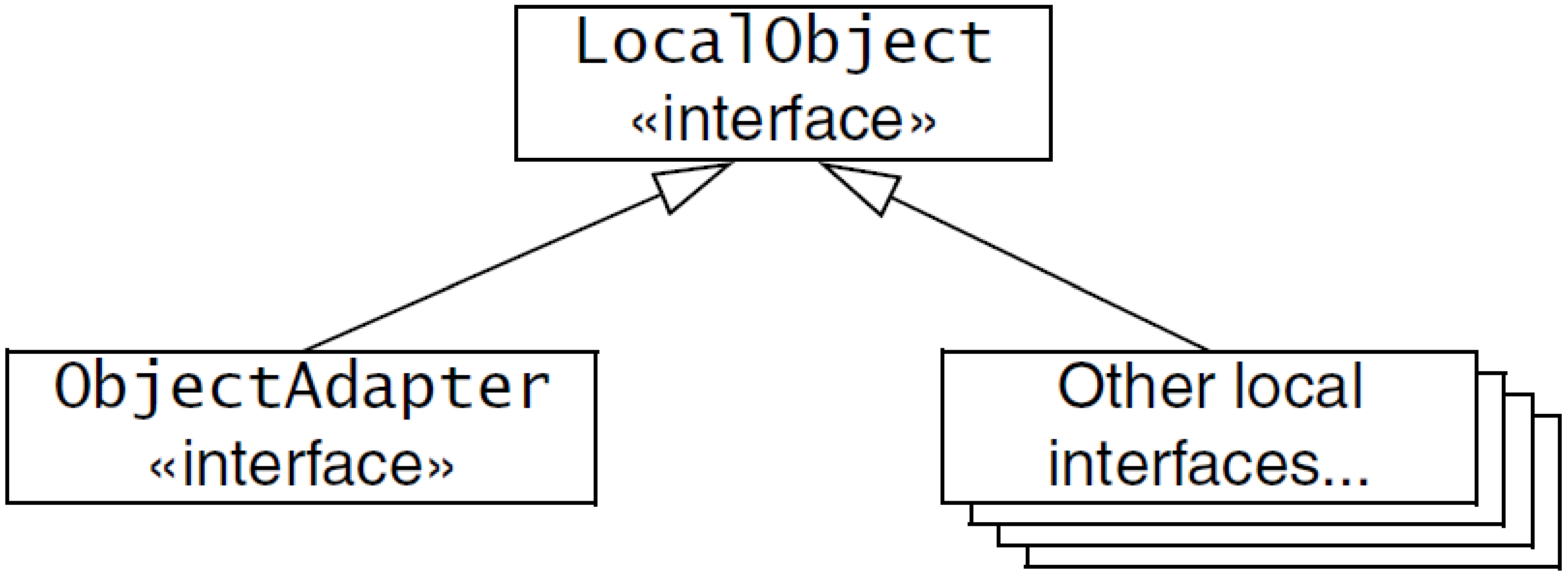
- In order to access certain features of the Ice run time, you must use APIs that are provided by libraries.
  - Instead of defining an API that is specific to each implementation language, Ice defines its APIs in Slice using the local keyword.
  - The advantage of defining APIs in Slice is that a single definition suffices to define the API for all possible implementation languages.
    - The actual language-specific API is then generated by the Slice compiler for each implementation language.
  - Types that are provided by Ice libraries are defined using the Slice local keyword.

```
module Ice {
  local interface ObjectAdapter {
    // ...
  };
};
```

- Any Slice definition (not just interfaces) can have a local modifier.
  - If the local modifier is present, the Slice compiler does not generate marshaling code for the corresponding type.
    - A local type can never be accessed remotely because it cannot be transmitted between client and server.

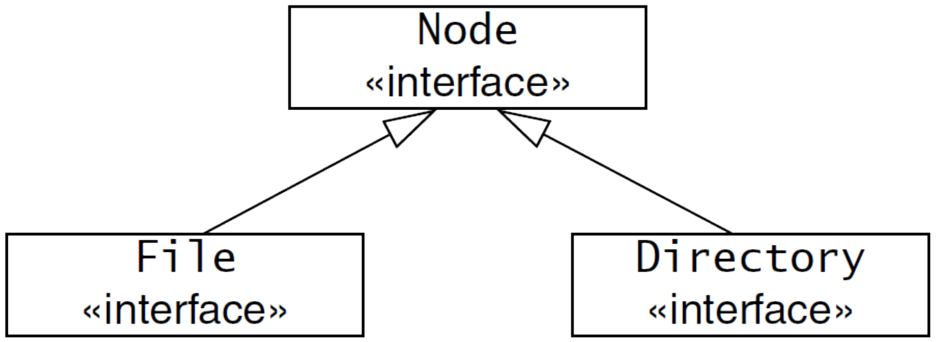
Local Types

- Local interfaces and local classes do not inherit from **`Ice::Object`**.
- Instead, local interfaces and classes have their own, completely separate inheritance hierarchy. At the root of this hierarchy is the type **`Ice::LocalObject`**.
- You cannot pass a local interface where a non-local interface is expected and vice-versa.



Slice for a Simple File System

```
module Filesystem {
  interface Node {
    idempotent string name();
  };
  exception GenericError {
    string reason;
  };
  sequence<string> Lines;
  interface File extends Node {
    idempotent Lines read();
    idempotent void write(Lines text) throws GenericError;
  };
  sequence<Node*> NodeSeq;
  interface Directory extends Node {
    idempotent NodeSeq list();
  };
};
```


Node
«interface»
File
«interface»
Directory
«interface»