

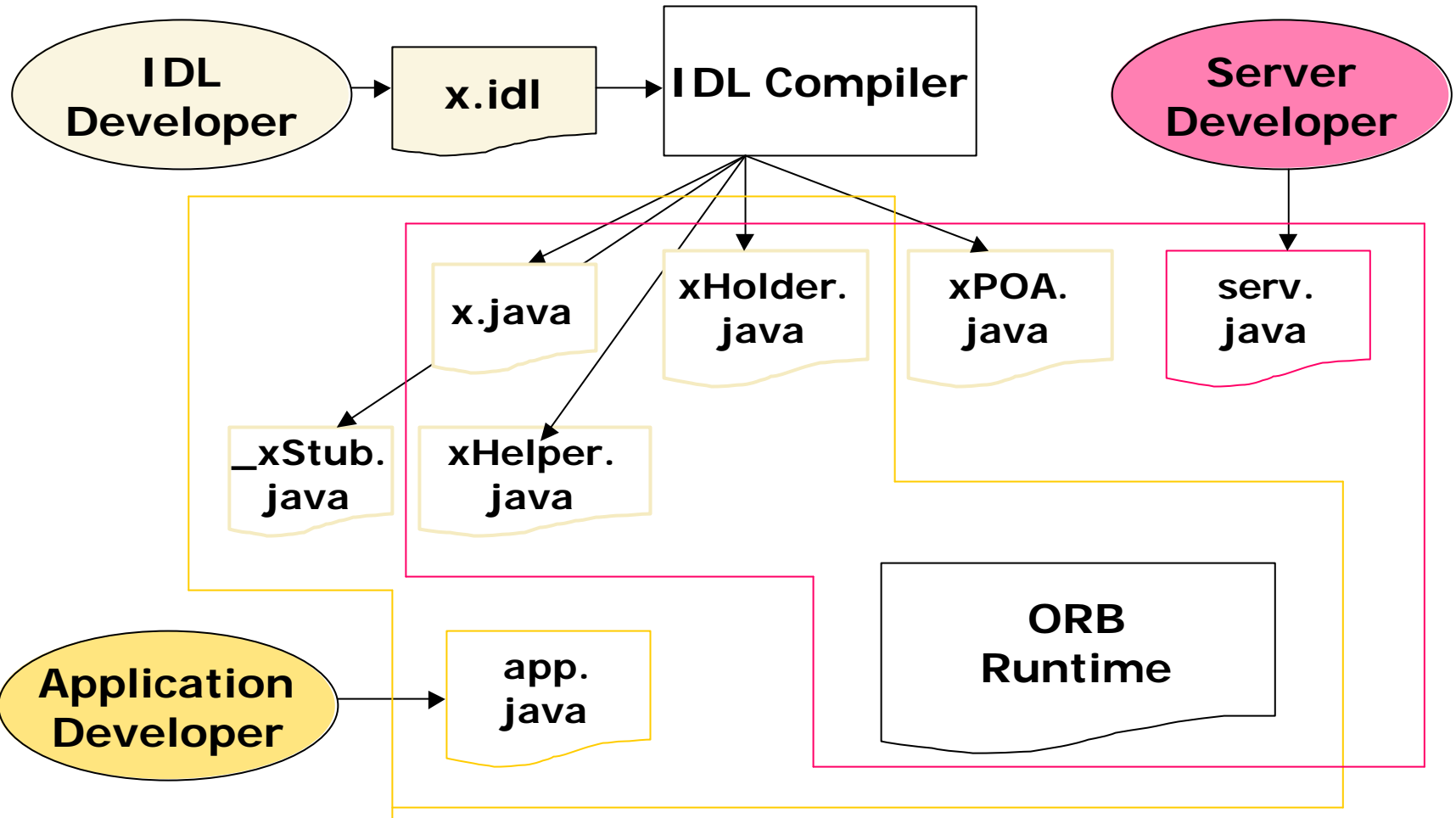
# Architecture of the CORBA Component Model

Interface Definition Language

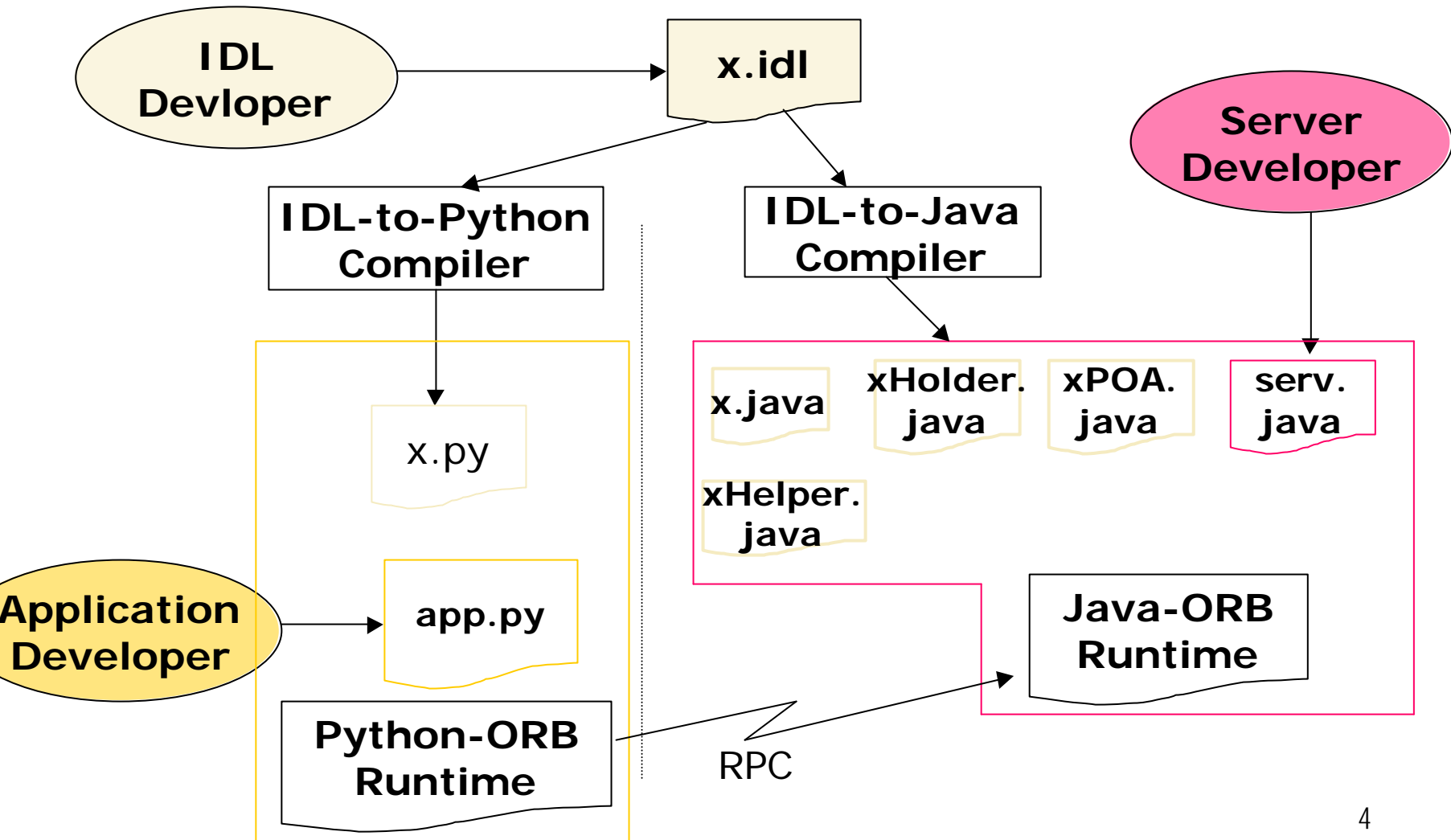
# Introduction

- IDL specifications separate language-independent **interfaces** from language-dependent implementations.
- IDL defines an **interface contract** between client and server.
- Language-independent IDL specifications are translated with an **IDL compiler** into APIs of the programming language.
- IDL is purely **declarative** (no actions, no statements about object state).
- IDL declarations are similar to Java interface definitions and to abstract classes.
- Data exchange between client and server is limited to the data types declared in IDL.

# IDL Translation (Java)



# IDL Translation (multiple languages)



# IDL Source Files

- IDL files must end in .idl
- IDL is free-format: Line breaks and white space have no significance
  - Except for preprocessor
- Sources are processed by the preprocessor (**#include**, **#define**, ...)
- Order of declarations is irrelevant
  - Definition must occur before use
  - Forward declarations

# Comments and Keywords

- IDL supports C and C++ style comments

/\*

\* A C comment

\*/

// A C++ comment

- IDL keywords are all lower-case (e.g. **interface**), except for **TRUE**, **FALSE**, **Object**, and **ValueBase**.

# Identifiers

- IDL identifiers can contain letters (A-Za-z), digits, and the underscore, e.g.

**Thermometer, nominal\_temp**

- IDL identifiers start with a letter. A leading underscore is allowed and ignored: **set\_temp** and **\_set\_temp** are the same.
- Identifiers are case insensitive; **max** and **MAX** are the same identifier. The spelling must be consistent.
- Identifiers which are keywords in programming languages should be avoided (e.g. **class, package, template**).

# Builtin Types

Various integral and floating point types (inherited from C)

Type	Size	Value Range
<b>short</b>	$\geq 16$ bits	$-2^{15} \dots 2^{15}-1$
<b>unsigned short</b>	$\geq 16$ bits	$0 \dots 2^{16}-1$
<b>long</b>	$\geq 32$ bits	$2^{31} \dots 2^{31}-1$
<b>unsigned long</b>	$\geq 32$ bits	$0 \dots 2^{32}-1$
<b>long long</b>	$\geq 64$ bits	$2^{63} \dots 2^{63}-1$
<b>unsigned long long</b>	$\geq 64$ bits	$0 \dots 2^{64}-1$
<b>float</b>	$\geq 32$ bits	IEEE single precision
<b>double</b>	$\geq 64$ bits	IEEE double precision
<b>Long double</b>	$\geq 79$ bits	IEEE extended precision



# Builtin Types (2)

- CORBA 2.1 adds fixed-point types:

```
typedef fixed<9,2> Total;           // up to 9 999 999,99
                                     // Precision 0,01
typedef fixed<9,4> InterestRate;   // up to 99 999,9999
                                     // Precision 0,0001
typedef fixed<31,0> BigInt;        // up to 10^31-1
```

- Fixed-point types have up to 31 digits
- No rounding effects in decimal system
- Computations use 62 digits

# Builtin Types (3)

IDL has two character types, `char` and `wchar`.

- `char` is an 8-bit type, `wchar` is wider (2 to 6 bytes).
- The standard code for **`char`** is ISO Latin-1; for **`wchar`**, it is 16-bit Unicode.

Accordingly, there are two string types, `string` and `wstring`.

- Strings may contain arbitrary characters except for NUL.
- Strings can be limited in size

```
typedef string      City;  
typedef string<3>  Abbreviation;  
typedef wstring    Stadt;  
typedef wstring<3> Abkueerzung;
```

# Builtin Types (4)

- The IDL type **octet** is an 8-bit type which is transmitted without change.  
It is used to transmit binary data.
- **boolean** is a type with the values **TRUE** and **FALSE**.
- **any** is a universal type:
  - A value of type **any** can carry arbitrary values of other types, e.g. **boolean**, **double**, or user-defined types.
  - Values inside the **any** are type safe: extracting a value as a different type is not allowed.
  - **any** provides introspection: Given an **any** value, the type of the encapsulated value can be obtained.

# Type Definitions

- Using a typedef, new names for an existing type can be introduced:

```
typedef short      YearType;
```

```
typedef short      TempType;
```

```
typedef TempType   TemperatureType
```

- Every type should have an application-specific name; these names should then be used consistently.
- Skilled use of **typedef**s increases readability.
- Unnecessary type aliases should be avoided; they are confusing and cause incompatibilities in some languages.

# Enumeration types

IDL allows the definition of enumerations:

```
enum Color {red, green, blue, black, mauve, orange};
```

- **Color** is a type of its own; no further typedef is needed.
- The type name must be provided.
- The enumerators are in the enclosing namespace and must be unique there:

```
enum InteriorColor {white, beige, grey};
```

```
enum ExteriorColor {yellow, beige, grey}; //Error!
```

- One cannot assign ordinals to enumerators:

```
enum Wrong{ red = 0, blue = 8};
```

# Structures

Structures are types with fields of arbitrary other types (including other user-defined types, excluding recursive types)

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

- A structure must contain atleast one field.
- The structure name must be provided.
- Member names must be unique within the structure.
- Structures form namespaces.
- Typedefs for structures should be avoided.

# Unions

IDL supports “discriminated unions” with arbitrary fields

```
union ColorCount switch Color{  
case red:  
case green:  
case blue:  
    unsigned long    num_in_stock;  
case black:  
    float            discount;  
default:  
    string           order_details;  
};
```

# Unions (2)

- A union must have at least one field.
- The type name must be provided.
- Unions form namespaces with unique member names.

## Union Usage Guidelines:

- **char** should not be used as the discriminator type.
- Unions should not be used to model “type casts”.
- There should be only one union field per case label.
- The default branch should not be used.
- Unions should be used sparingly.



# Arrays

IDL supports one- and multi-dimensional array with arbitrary element type.

```
typedef Color ColorVector[10];
```

```
typedef string IdTable[10][20];
```

- Using a typedef here is mandatory;  
Color ColorVector[10];  
is ill-formed
- All dimensions must be provided;  
typedef string OpenTable[][20];  
is also ill-formed.
- Exercise caution when passing array indices across address spaces!

# Sequences

Sequences are vectors of variable length.

- Sequences can be bounded or unbounded.

```
typedef sequence<Color> Colors;
```

```
typedef sequence<long, 100> Numbers;
```

- The bound must be a positive integral number.
- Sequences must be defined in a **typedef**.
- The element type can be an arbitrary type (including a recursive type)

```
typedef sequence<Node> ListOfNodes;
```

```
typedef sequence<ListOfNodes> TreeOfNodes;
```

- Sequences can be empty.

# Arrays or sequences?

Arrays and sequences are similar, hence a few recommendations:

- If you have a fixed-size list of values, and all values are always present, use an array.
- If you have a variably-sized set of things, use a sequence.
- Use character arrays for strings of fixed size.
- Use sequences for sparse matrices (with (i, j, value) triples)

# Recursive Types (historical)

- IDL has no pointers, yet it supports recursive data structures:

```
struct Node{  
    long value;  
    sequence<Node> children;  
};
```

- Structures themselves cannot be recursive
- CORBA 2.3: **valuetypes**

# Constants and Literals

Constants can be defined for arbitrary builtin types (except **any**), and for enumerations

```
const long FAMOUS_CONST=42;
const double SQRT_2 = 1.414213;
const char FIRST = '42';
const string GREETING = "Gooday, mate!";
const octet LSB_MASK = 0x01;

typedef fixed<6,4> ExchangeRate;
const ExchangeRate UNITY = 1.0D;

enum Color {ultra_violent, burned_hombre,
            infra_dead};
const Color NICEST_COLOR = ultra_violent;
```

# Constant Expressions

- Arithmetic operators

+ - \* / %

- Bit-wise operators

& | ^ << >> ~

# Interfaces

- Interfaces (Schnittstellen) define object types:

```
interface Thermometer{  
    string    get_location();  
    void      set_location(in string loc);  
};
```

- Invocation of an operation for an instance sends an RPC call to the server implementing the instance
- Interfaces define the “public” Interface. There are no private/protected parts.
- Interfaces have no data members.
- Interfaces define the smallest and only granularity of distribution (*unit of distribution*). Everything remotely accessible has an interface.

# Interface Syntax

- Interface definitions may include exceptions, attributes, operations and type definitions

```
interface Haystack{  
    exception NotFound{ unsigned long  
        num_straws_searched;};  
    const unsigned long MAX_SIZE = 1000000;  
    readonly attribute unsigned long num_straws;  
    typedef long Needle;  
    typedef string Straw;  
    void add(in Straw s);  
    boolean remove(in Straw s);  
    boolean find(in Needle n) raises(NotFound);  
};
```



# Interface Semantics

- Interfaces are types and can be used as parameters

```
interface FeedShed{  
    void          add(in Haystack s);  
    void          eat(in Haystack s);  
};
```

- Parameters of type **Haystack** are object reference parameters.
- Passing of objects always happens by reference.
- The object remains in its original location, only the reference is passed.
- Usage of a reference again happens by RPC calls.
- Nil reference: no object

# Syntax of Operations

Every operation definition contains

- An operation name
- A return type (possibly **void**)
- Zero or more parameters

Optionally, a operation definition contains

- A **raises** declaration
- A **oneway** attribute
- A **context** clause

IDL provides no operation overloading; operation names must be unique within their interface.

# Operations: Example

```
interface NumberCruncher{  
    double square_root(in double operand);  
    void square_root2(in double operand,  
                     out double result);  
    void square_root3(inout double op_res);  
};
```

- Parameter have a directionality attribute:
  - **in**: Parameter is sent from the client to the server
  - **out**: Parameter is returned from the server to the client
  - **inout**: Parameter is sent from the client to the server, possibly modified at the server, and sent back to the client.

# User-defined Exceptions

- Possible exceptions are declared with the **raises** clause

```
exception Failed{};
exception RangeError{
    unsigned long    min_val;
    unsigned long    max_val;
};
interface Unreliable{
    void can_fail()raises(Failed);
    void can_also_fail(in long l)raises(Failed,
    RangeError);
};
```

- Exceptions are defined like structures but need not have any members
- Exception definitions cannot be nested, inherited, or used as data types

# Exceptions: Usage Guidelines

- Exceptions should only be used in the exceptional case.
- Exceptions should only contain *useful* information.
- Exceptions should contain *precise* information.
- Exceptions should contain *complete* information.

# System Exceptions

CORBA defines 39 system exceptions.

- Any operation can raise a system exception.
- System exceptions must not be mentioned in **raises** clauses.
- All system exceptions have the same parameters:

```
enum completion_status{
    COMPLETED_YES, COMPLETED_NO, COMPLETED_MAYBE
};
exception <SystemExceptionName>{
    unsigned long      minor;
    completion_status  completed;
};
```

# Oneway Operations

Operations can be defined as **oneway**:

```
interface Events{  
    oneway void send(in EventData data);  
};
```

Semantic restrictions for oneway operations:

- The return type must be void.
- They must not have **out** or **inout** parameters.
- There must be no **raise** clause.

Oneway operations provide best-effort send-and-forget semantics.

Oneway calls can be lost and can be delivered synchronously or asynchronously.

In CORBA 2.3, oneway is not fully portable (2.4: async messaging spec)

# Contexts

Operations can optionally provide a **context** clause.

```
interface Poor{  
    void doit()  
    context( "USER" , "GROUP" , "X*" );  
};
```

When invoking "doit", the context variables USER, GROUP, and all which start with X are transmitted.

Contexts are similar to Unix environment variables.

Many ORBs don't implement contexts incorrectly, so using them should be avoided.



# Attributes

Interfaces may contain attributes:

```
interface Thermostat{  
    readonly attribute short temperature;  
    attribute short nominal_temp;  
};
```

- Attributes imply an **operation pair**: a read operation and a write operation (readonly: only the read operation)
- Attributes define neither state nor members; they are merely a short-hand notation.
- Attributes must not throw exceptions (up to CORBA 2.4), must not be **oneway**, and must not have contexts.
- Attributes should be read-only.

# Modules

- Modules are namespaces, similar to C++ namespaces

```
module M {  
  //...  
  module L {  
    //...  
    interface I{ /*...*/ };  
    //...  
  };  
  //...  
};
```

- Modules help avoiding naming conflicts.
- Modules can contain arbitrary IDL constructs.
- Modules can be extended (reopening).

# Forward Declarations

Forward declarations of interfaces allow the definition of interfaces that mutually depend on each other:

```
interface Husband;  
interface Wife{  
    Husband get_spouse();  
};  
interface Husband{  
    Wife get_spouse();  
};
```

The name in the redefinition must be *simple*:

```
interface MyModule::SomeInterface; //Syntax error!
```

# Inheritance

IDL allows the inheritance of interfaces (and valuetypes):

```
interface Thermometer{  
    typedef short TempType;  
    readonly attribute TempType temperature;  
};
```

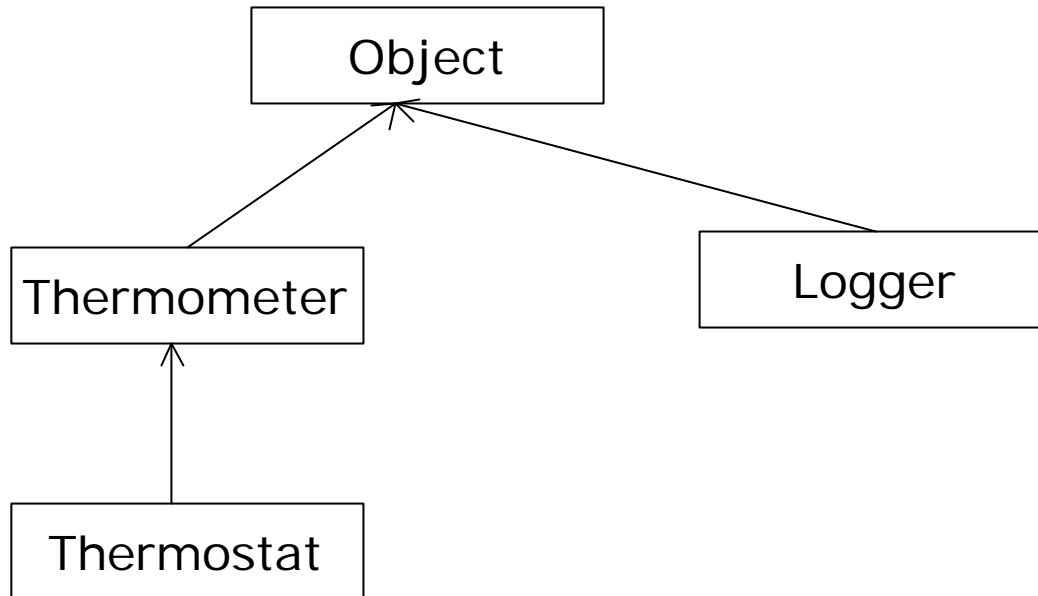
```
interface Thermostat: Thermometer{  
    void set_nominal_temp(in TempType t);  
};
```

- Polymorphism: Specialized interfaces can be passed where base interfaces are expected.

```
interface Logger{  
    long add(in Thermometer t, in unsigned short  
    interval);  
    void remove(in long id);  
};
```

# Inheritance from Object

- All interfaces implicitly inherit from Object



- When an operation has an Object argument, arbitrary objects can be passed

# Redefinition and Inheritance

Types, constants, and exceptions can be redefined in the specialization

```
interface Thermometer{  
    typedef long IDType;  
    const IDType TID = 5;  
    exception TempOutOfRange{};  
};  
interface Thermostat:Thermometer{  
    typedef string IDType;  
    const IDType TID = "Thermostat";  
    exception TempOutOfRange { long temp; };  
};
```

# Inheritance Restrictions

- Operations and Attributes must not be redefined in the specialization.

```
interface Thermometer{  
    attribute long temperature;  
    void initialize();  
};  
  
interface Thermostat:Thermometer{  
    attribute long temperature; //error  
    void initialize(); //error  
};
```

# Multiple Inheritance

Multiple Inheritance (including multiple usage of the same interface) is allowed.

```
interface Sensor{  
    // ...  
};  
interface Thermometer:Sensor{  
    //...  
};  
interface Hygrometer: Sensor{  
    //...  
};  
interface HygroTherm: Thermometer,Hygrometer{  
};
```



# Rules for Multiple Inheritance

- Operations with the same name must not originate from different base interfaces.
- Types with the same name may be inherited but need to be qualified on usage.

# Value Types

Motivation:

- Communication with objects occurs in a request/response protocol.
- Objects cannot be transmitted as values.
- Structured values are limited: no graphs, no methods.

CORBA and fine-grained objects:

- Example: Interface for a database
- Every record is an object?
- Every field is an access operation?
- Every access is a round-trip communication.

# Value Types (2)

## History:

- RFP orbos/96-06-14:
  - “pass-by-value” RFP
  - “objects-by-value” RFP
- Adopted specification:
  - orbos/98-08-18
- No transmission of objects by value
- valuetype: new metatype

# Value types (3)

Application areas:

- Extensible structures
- Recursive and cyclic structures
- Complete implementation of Java RMI with CORBA (RMI over IIOP)
  - Java-to-IDL-Mapping defines creation of IDL interfaces

# Value Types (4)

```
valuetype Person{
    public string name;
    private short birth_year;
    short age();
};

valuetype Employee:Person{
    public string department;
};
```

# Value Types (5)

Properties of a value type:

- Data attributes
- Methods
- Inheritance
- Type name
- Value factories
- Visibility/Access Control

The type Person has two data fields:

- Transmission of values in RPC

Method definitions require local implementations

Employee has an additional type:

- Receiving ORB must know complete structure

# Value Types (6)

Usage in object operations:

```
module M{
    valuetype Person{...};
    interface Department{
        void hire(in Person p);
        Person get(in string name);
        void fire(in string name);
    };
};
```

# Value Types (7)

- Simple Inheritance for concrete valuetypes
- Abstract valuetypes: Only methods allowed; multiple inheritance is possible

```
abstract valuetype Printable{  
    void print();  
};
```

```
valuetype Employee:Person, Printable{  
    public string Department;  
};
```

- Truncation: Receiving ORB can omit unknown fields

```
valuetype Manager: truncatable Employee{  
    public Department in_charge_of;  
};
```

- Universal base type ValueBase



# Value Types (8)

- Recursive Types:

```
valuetype Tree{  
    private Tree left,right;  
    void traverse(in Action todo);  
};
```

```
valuetype StringTree:Tree{  
    public string content;  
};
```

- New keywords: valuetype, truncatable, private, public, factory, custom, abstract, supports

# Value Types (9)

Value boxes:

- Valuetypes allow recursive/cyclic structures:
  - Mapped to objects/pointers
  - Need null value
- Valueboxes allow to wrap "old" types as a value

```
valuetype LongBox long;
```

# Value Types (10)

- Methods of valuetype:
  - Always local
  - Not ORB-mediated (programming language semantics applies)
- Passing values to object operations:
  - Receiver gets copy (even in a co-located call)
  - Copy is isomorphic to the original
- Factories
  - Receiving ORB must create instance
  - Method implementation is specific to the application
  - Application must register a value factory

# Value Types (11)

- More features (not presented):
  - Custom marshalling
  - Factory operations
  - Supported interfaces (passing a value by reference depending on context)