CORBA

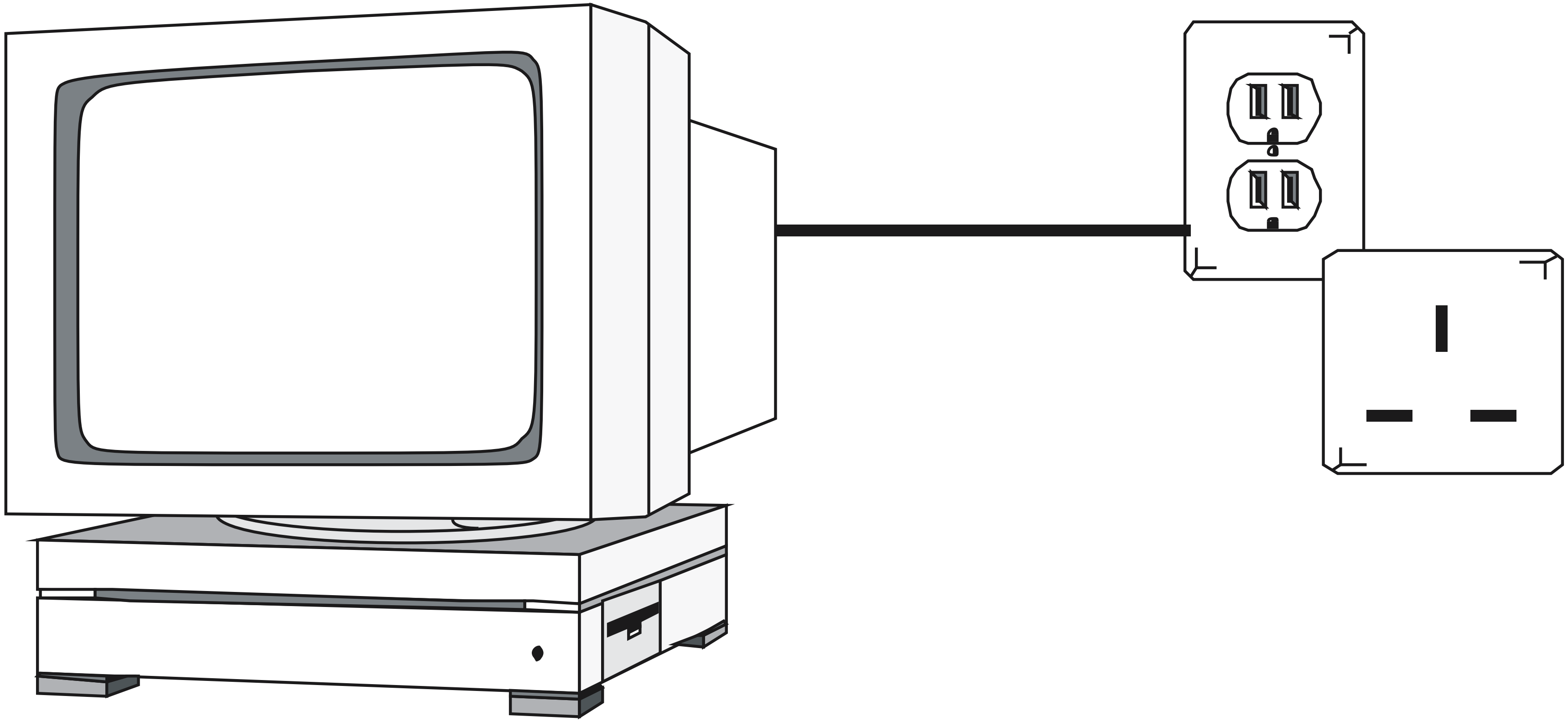
Common Object Request Broker Architecture

CORBA

- Common ((Object Request) Broker) Architecture
- Standardization: Object Management Group (http://www.omg.org)
- Current specification version: 3.1
- M. Henning, S. Vinoski: Advanced CORBA Programming with C++, Addison-Wesley 1999

Global Information Appliance

- Connecting the device to the computer network as easy, as to the power supply network



OMG Range of Interest

- Application integration
  - Distributed processing
- Systems processing information coming from diferent sources
  - Heterogeneous
  - Networked
  - From different vendors

Previous Approaches

- Too low-level
  - Good components, but not at the level interesting for the application designer
- Lack of higher-level standardization

Creation of OMG Standards

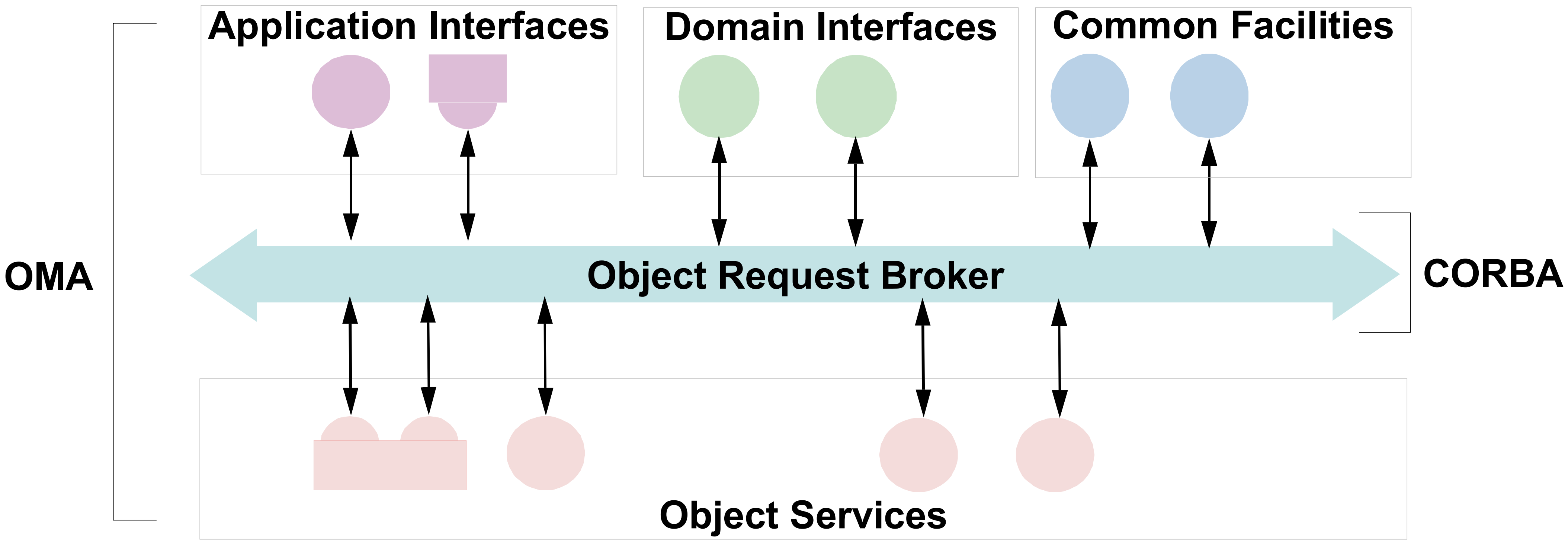
- OMG choses interfaces
  - From competing industry proposals
- OMG publishes interfaces
  - Freely available for everybody
- OMG controls interfaces
  - They are owned by OMG, it controls their development
- OMG cooperates with standard bodies
  - Specifications become standards

OMG Is Neutral

- Does not create nor sell implementations
- Does not test implementations
  - Test are performed by X/Open

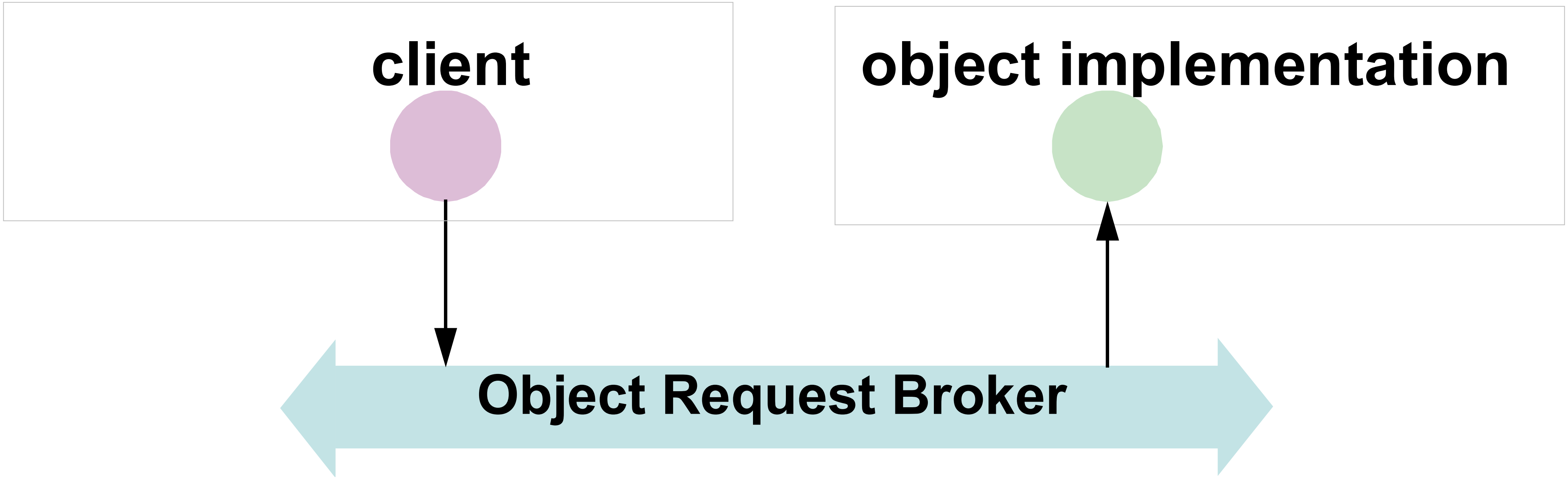
Object Management Architecture

- Object Request Broker + objects
  - Object services, Common Facilities, Domain Interfaces, Application Interfaces



Object Request Broker (ORB)

- Objects acquire access to other objects via ORB
  - ORB locates object implementations and deals with communication with them
  - Client and server can be written in different languages and run on different types of computers



Object Services (CORBA services)

- Basic services at the system level
  - Are objects
  - Have the interface specification
  - Different implementations possible
- Examples
  - Transactions
  - Concurrency
  - Events

Common Facilities(CORBA Facilities)

- Objects shared by applications
  - E-mail
  - Printing
- Allow cooperation of products from different vendors

Domain Interfaces

- Standard objects for different application domain
  - Geospatial data processing
  - System management

Application Interfaces

- Application-specific
  - Provided by independent software vendors
  - Provided by end user
- Not specified by OMG

Special Interest Groups

- Internet
- Realtime
- Electronic Commerce

CORBA Is Portable

- Platforms and operating systems
- Programming languages: C C++ SmallTalk Ada95 COBOL Java LISP PL/1 Python

CORBA – Programmer's Interface

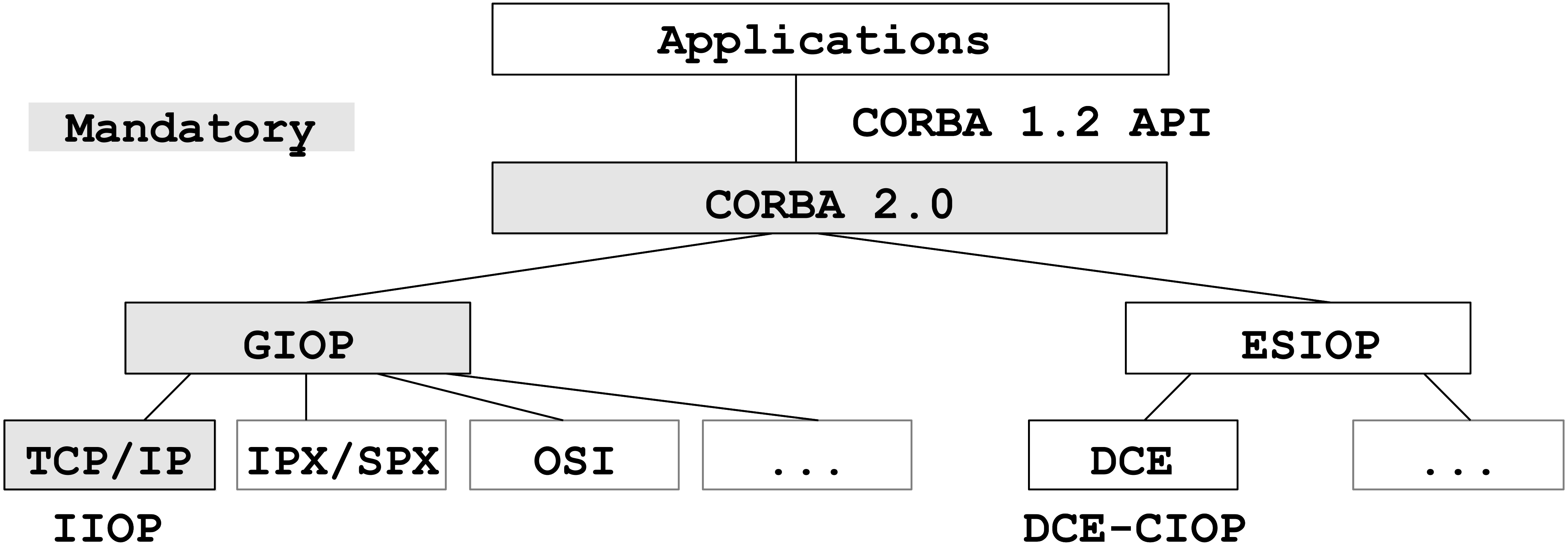
- CORBA is object-oriented
- Requires use of object-oriented programming principles
  - Not necessarily in the object-oriented language (e.g. C)

Portability, Heterogenity and Interoperability

- Does not require the same hardware platform and the same programming language in the entire application
  - Client in Java on the palmtop
  - Server in COBOL on the mainframe
- Allows for later change of platform and programming language
- Requires interoperability between implementations from different vendors

Protocols Used in CORBA Standard

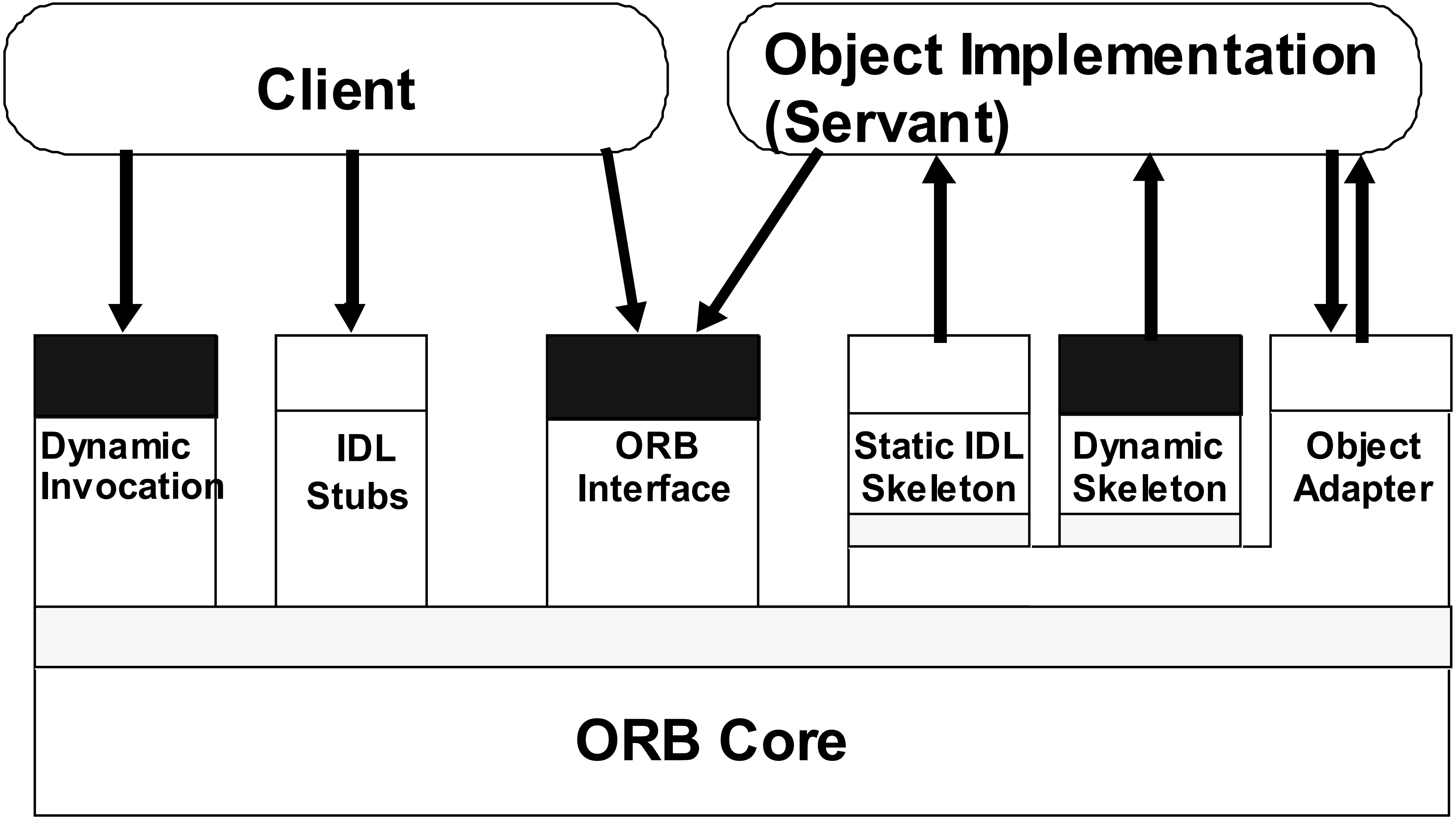
- IIOP (Internet Inter-ORB Protocol) is obligatory
- DCE-CIOP (Common Inter-ORB Protocol) is optional



Design Goals for GIOP and IIOP

- Wide availability
- Simplicity
- Scalability
- Low cost
- Generality
- Architecture neutrality

Control Flow at Operation Invocation



Client

Object Implementation (Servant)

Dynamic Invocation

IDL Stubs

ORB Interface

Static IDL Skeleton

Dynamic Skeleton

Object Adapter

ORB Core

■ Interface identical for all ORB implementations

■ There may be multiple object adapters

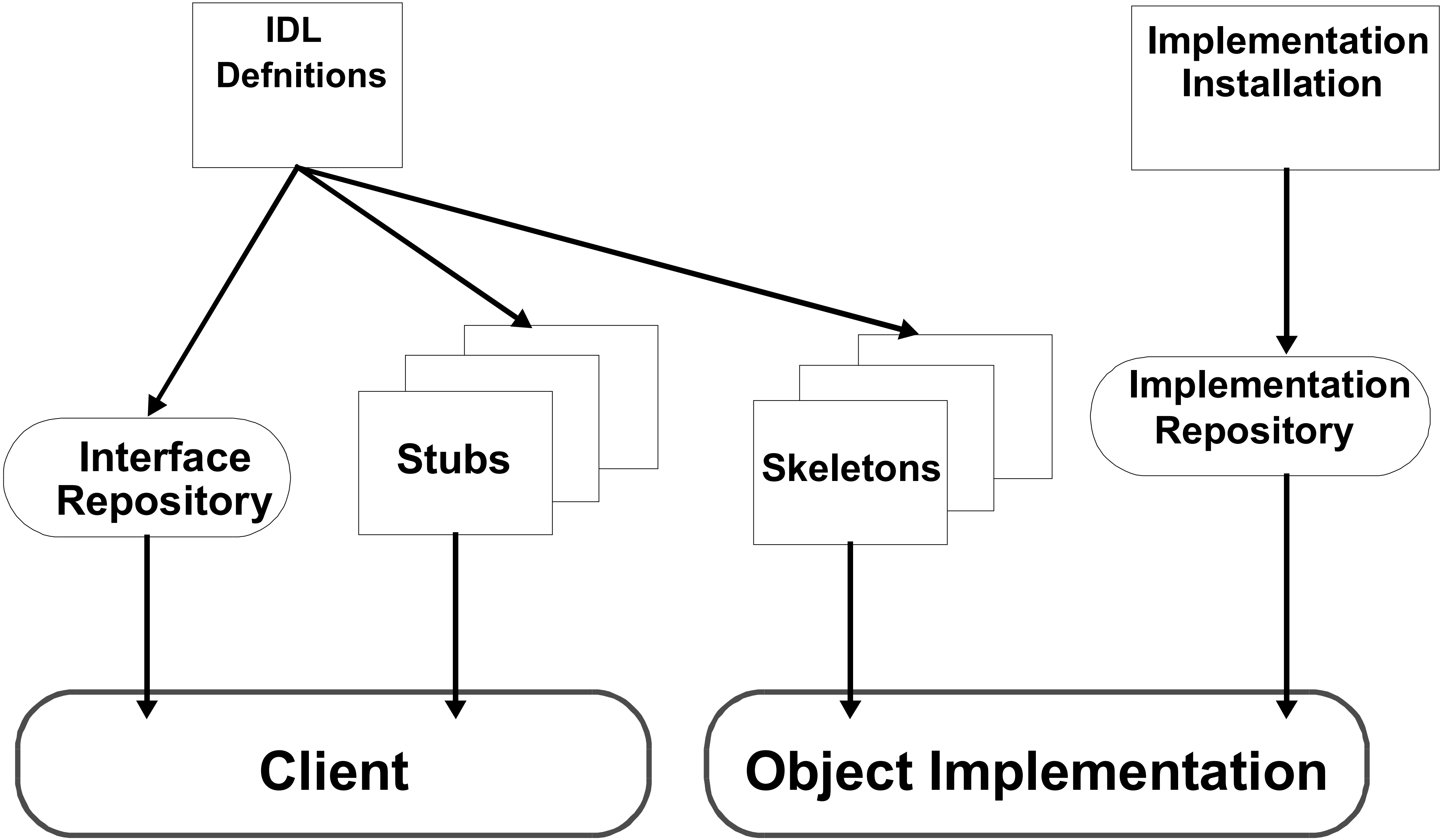
■ There are stubs and a skeleton for each object type

□ ORB-dependent interface

↑ Up-call interface

↓ Normal call interface

Interface Repository and Implementation Repository



Call Types

- Synchronous
  - Blocking until the response is received
- Deferred synchronous
  - Send request and check, if response has arrived
- Oneway
  - Does not guarantee that request is sent
- Asynchronous (CORBA 3.0)
  - Callback or polling

OMG Interface Definition Language

```
interface Employee {
  long number();
};

interface  EmployeeRegistry {
  Employee lookup(in long emp_number);
};
```

Interface Inheritance

```
interface Printer {
  void print();
};

interface  ColorPrinter: Printer {
  enum ColorMode {BlackAndWhite, FullColor };
  void set_color(in ColorMode mode);
};
```

- All the interfaces are implicitly inherited from the Object interface

Call and Dispatch

- Static call and dispatch
  - Stubs and skeletons
- Dynamic call and dispatch
  - Dynamic Invocation Interface
  - Dynamic Skeleton Interface

Object Adapters

- Create object references, making addressing of objects possible
- Ensure object incarnation by the server
- Receive requests from ORB at the server side and dispatch them to the appropriate servants
- Portable Object Adapter

Operation Invocation by ORB

- Locates the target object
- Activates server, if necessary
- Sends the operation arguments to the server
- Waits for completion of the operation
- Returns the **`out`**, **`inout`** parameters and the return value to the client at successful execution of operation
- Returns an exception at operation failure

Operation Invocation Features in CORBA System

- Location transparency
- Server transparency
- Language independence
- Implementation independence
- Operating system independence
- Protocol independence
- Transport layer independence

Semantics of Object References

- Every reference identifies exactly one object
- Many references can refer to the same object
- References can be empty (not point to any object)
- References can point nonexistent object (dangling references)
- References are opaque for the client
- References are strongly typed
- References support late binding
- References can be persistent
- References can be used by different ORBs

Obtaining References

- As the result of operation
- Using standard services, line Naming Service or Trading Service
- By conversion to text string and write/read from the file
- By other methods (e-mail, web page)

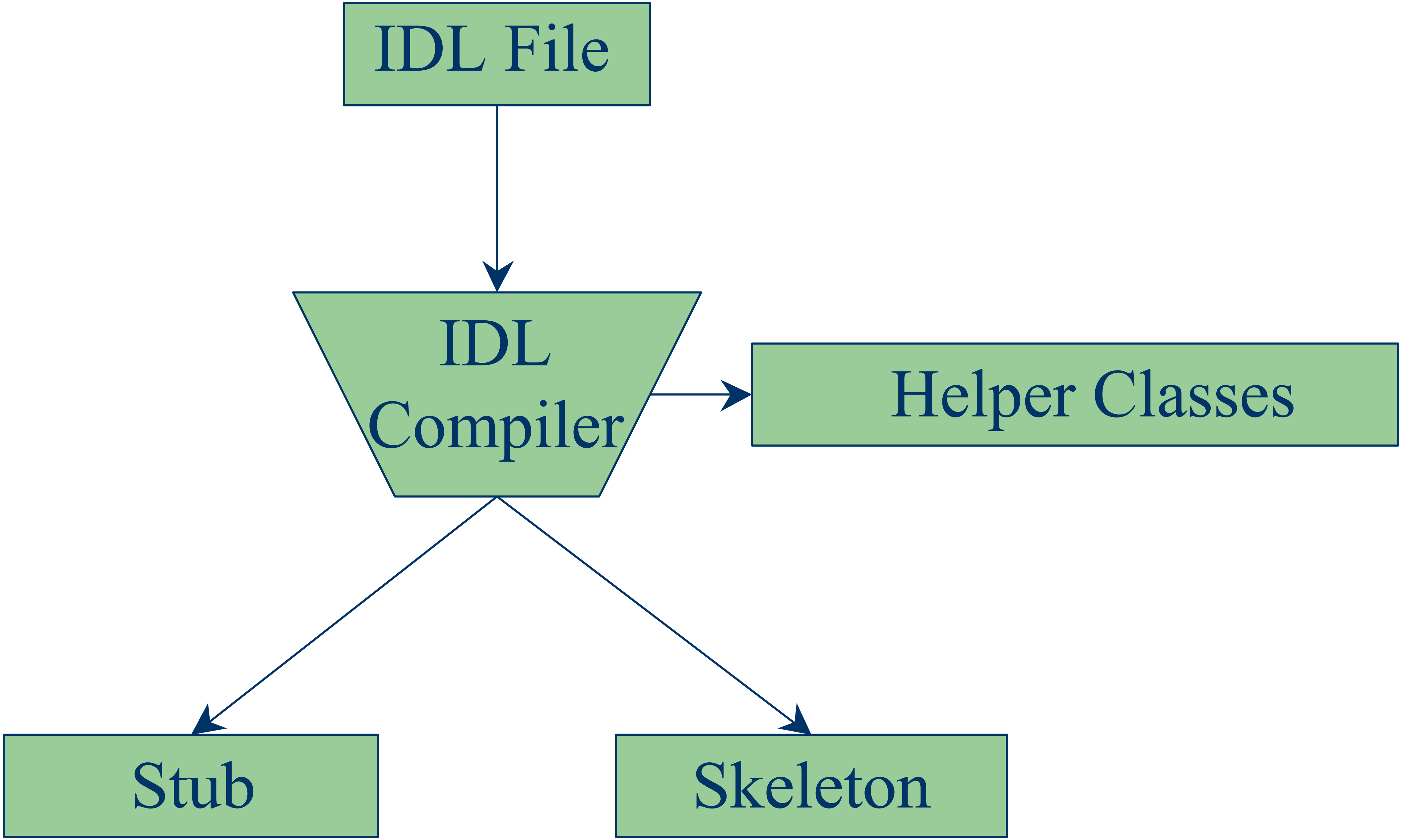
Interoperable Reference (IOR)

- Repository ID
  - Standardized
- Endpoint Info
  - Standardized
- Object Key
  - Not standardized

Writing CORBA Applications

- Define application objects and IDL interfaces
- Compile IDL definitions to stubs and skeletons
- Declare and implement servant classes in C++, incarnating the objects
- Write the main() function in the server
- Compile and link the server
- Write, compile and link the client

Writing CORBA Applications



Simple CORBA Application

```
struct TimeOfDay {
short   hour;   // 0 - 23
short   minute;// 0 - 59
short   second;// 0 - 59
};
interface Time {
    TimeOfDay get_gmt();
};
```

```
omniidl -bcxx time.idl
```

- time.hh
- timeSK.cc

```
//time.hh
class _impl_Time :
 public virtual omniServant
{
public:
 virtual ~_impl_Time();
 virtual TimeOfDay get_gmt() = 0;
};

class POA_Time :
 public virtual _impl_Time,
 public virtual
   PortableServer::ServantBase
{
public:
 virtual ~POA_Time();
 Time_ptr _this();
}
```

CORBA Server

```
//server.hh
#include "time.hh"

class Time_impl : public virtual
POA_Time {
public:
    virtual TimeOfDay get_gmt()
     throw(CORBA::SystemException);
};
```

```
//server.cc
#include <time.h>
#include <iostream>
#include "server.hh"
using namespace std;

TimeOfDay Time_impl::get_gmt()
throw(CORBA::SystemException)
{
   time_t time_now = time(0);
   struct tm * time_p =
                gmtime(&time_now);

   TimeOfDay tod;
   tod.hour = time_p->tm_hour;
   tod.minute = time_p->tm_min;
   tod.second = time_p->tm_sec;

   return tod;
}
```

CORBA Server (contd.)

```
// server.cc
int main(int argc, char * argv[]) {
try {
 // Initialize orb
 CORBA::ORB_var orb =
 CORBA::ORB_init(argc, argv);
 // Get reference to Root POA.
 CORBA::Object_var obj = orb ->
 resolve_initial_references
 ("RootPOA");
 PortableServer::POA_var poa =
 PortableServer::POA::_narrow(obj);
 // Activate POA manager
 PortableServer::POAManager_var
       mgr = poa->the_POAManager();
 mgr->activate();
 // Create an object
 Time_impl time_servant;
```

```
// Write its stringified
// reference to stdout
 Time_var tm =
 time_servant._this();
 CORBA::String_var str =
        orb->object_to_string(tm);
 cout << str << endl;
 // Accept requests
 orb->run();
}
catch (const CORBA::Exception &){
 cerr << "Uncaught CORBA"
 "exception" << endl;
 return 1;
}
return 0;
}
```

Compilation and Running

```
export OMNIHOME=/usr/local/omniORB-4.0.0

g++ -c -O2 -D__OMNIORB4__ -D_REENTRANT -I$OMNIHOME/include
-D__OSVERSION__=2 -D__linux__ -D__x86__ -o timeSK.o timeSK.cc

g++ -c -O2 -D__OMNIORB4__ -D_REENTRANT -I$OMNIHOME/include
-D__OSVERSION__=2 -D__linux__ -D__x86__ -o client.o client.cc

g++ -o server -L$OMNIHOME/lib timeSK.o client.o -lomniORB4
-lomnithread -lpthread

./server
IOR:010000000d00000049444c3a54696d653a312e3000000000010000000000
000064000000010102000e0000003231322e3234342e38362e343500cf800e00
0000fec16fa93d0000346b000000000000000200000000000000080000000100
000000545441010000001c000000010000000100010001000000010001050901
01000100000009010100
```

CORBA Client

```
//client.cc
#include <iostream>
#include <iomanip>
#include "time.hh"
using namespace std;

int main(int argc, char * argv[]) {
try {
// Initialize orb
CORBA::ORB_var orb =
CORBA::ORB_init(argc, argv);
// Check arguments
if (argc != 2) {cerr <<
"Usage: client IOR_string" << endl;
 throw 0; }
// Destringify argv[1]
CORBA::Object_var obj =  orb
        ->string_to_object(argv[1]);
 if (CORBA::is_nil(obj)) {
  cerr << "Nil Time reference" << endl;
  throw 0;
}
```

```
// Narrow
Time_var tm = Time::_narrow(obj);
if (CORBA::is_nil(tm)) {
cerr << "Argument is not a Time "
   "reference" << endl; throw 0; }
// Get time
TimeOfDay tod = tm->get_gmt();
cout << "Time in Greenwich is "
 << setw(2) << setfill('0') << tod.hour
 << ":" << setw(2) << setfill('0') <<
 tod.minute << ":" << setw(2) <<
 setfill('0') << tod.second << endl;
}
catch (const CORBA::Exception &) {
cerr << "Uncaught CORBA exception" <<
  endl;
return 1;
}
catch (...) {
 return 1;
}
return 0;
}
```

Compilation and Running

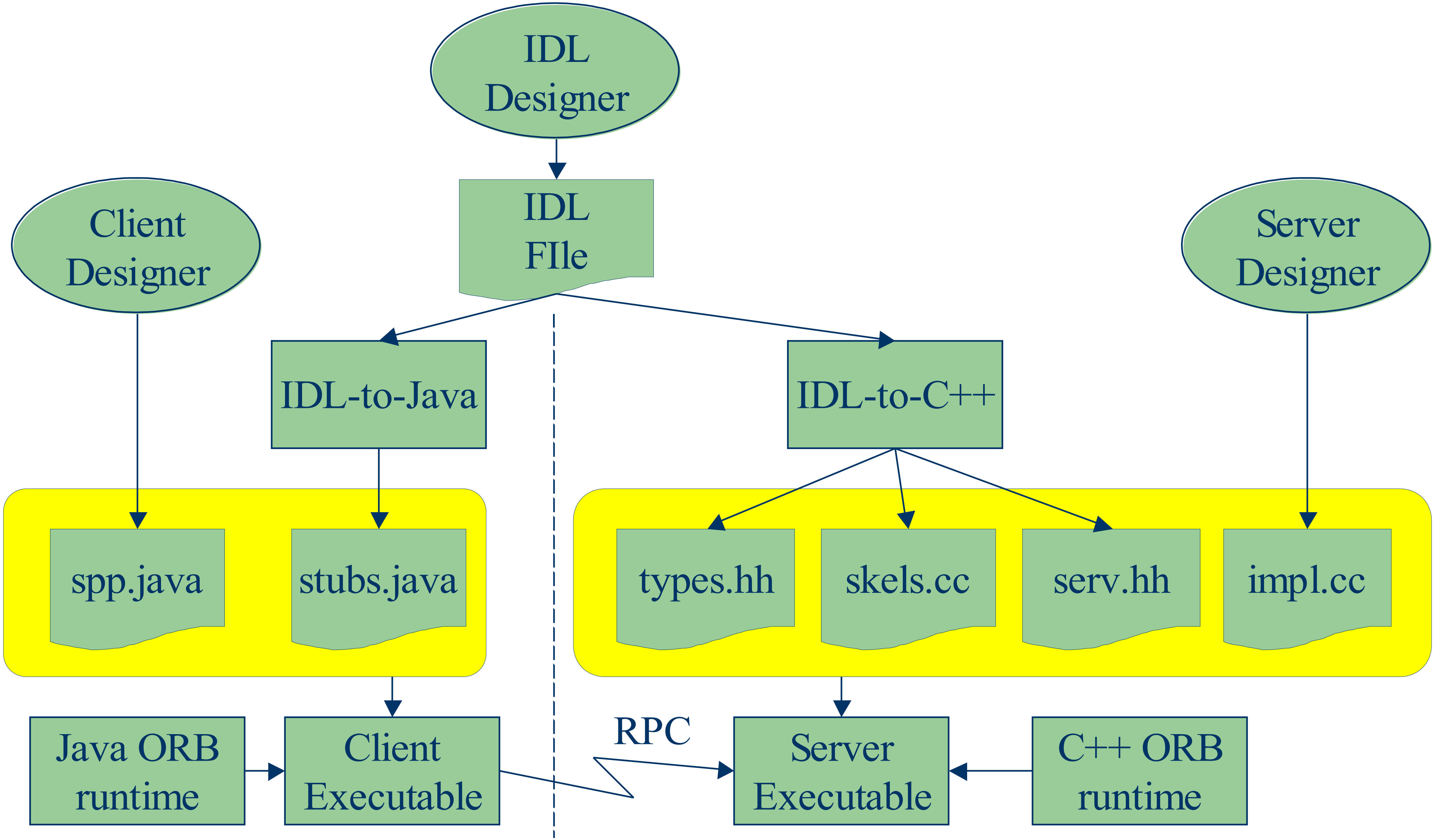
```
export OMNIHOME=/usr/local/omniORB-4.0.0

g++ -c -O2 -D__OMNIORB4__ -D_REENTRANT -I$OMNIHOME/include
-D__OSVERSION__=2 -D__linux__ -D__x86__ -o server.o server.cc

g++ -o server -L$OMNIHOME/lib timeSK.o server.o -lomniORB4
-lomnithread -lpthread

$ ./server > ior &
[1] 4587
$ ./client `cat ior`
$ Time in Greenwich is 21:06:54
$ kill %1
[1]+ Terminated          ./server &
$
```

Client and Server in Different Languages



IDL Source Files

- Extension .idl
- Free form
  - Tabulations, spaces, newlines do not matter
- Preprocessing
  - The same as C++ preprocessor
  - E.g. **`#define`**, **`#include`**
- Identifiers must be declared before use

Lexical Rules

- Comments
  - C-style /* */
  - C++-style //
- Keywords
  - Written in lowercase
  - Exceptions: **`Object`**, **`TRUE`**, **`FALSE`**
- Identifiers
  - Required consistent use of upper- and lowercase
  - Two identifiers differing only in character case not alowed

Basic IDL Types

Type	Range	Size
short	-2^{15} to $2^{15}-1$	≥16 bits
long	-2^{31} to $2^{31}-1$	≥32 bits
unsigned short	0 to $2^{16}-1$	≥16 bits
unsigned long	0 to $2^{32}-1$	≥32 bits
float	IEEE single precision	≥32 bits
double	IEEE double precision	≥64 bits
char	ISO Latin-1	≥8 bits
string	ISO Latin-1 except for ASCII NUL	variable
boolean	TRUE or FALSE	undefined
octet	0 to 255	≥8 bits
any	Identified at run-time	variable

The Any Type

- Type **`any`**
  - universal type
  - similar to **`void*`** or **`stdarg`**
  - better, because self-describing
  - strongly typed, misinterpretation more difficult

User-defined Types

- Named types - **`typedef`**
- Enumeration types - **`enum`**
- Structures - **`struct`**
- Unions - **`union`**
- Arrays
- Sequences - **`sequence`**
- Recursive types
- Constants and literals

Named Types

```
typedef short  YearType;
typedef short  TempType;
typedef TempType TemperatureType;
```

- The last declaration is bad style - unnecessary different name for the same type
- It is undefined, if **`TempType`** and **`TemperatureType`** are separate types, or can be used interchangeably - depends on the target language

Enumeration Types

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

- **`typedef`** not needed for enumeration types
- Type with at least 32-bit size
- Specific values for individual labels not defined
- Ensures, that the values are growing from left to right

Enumeration Types (contd.)

- Assigning specific values to the labels is illegal –

  ```
  enum Color { red = 0, green = 7 };
  ```

  is incorrect
- Labels belong to the surrounding namespace, so the following code is incorrect:

```
enum InteriorColor {white,beige,grey};
enum ExteriorColor {yellow,beige,green};
```

- Enumeration type cannot be empty

Structures

- Have one or more fields of any type, including user-defined types

```
struct TimeOfDay {
  short hour;
  short minute;
  short second;
};
```

Structures (contd.)

- Structure definition creates a separate namespace

```
struct Outer {
  struct FirstNested {
    long first;
    long second;
  } first;
  struct SecondNested {
    long first;
    long second;
  } second;
};
```

Structures (contd.)

- Almost the same, but more readable:

```
struct FirstNested {
  long first;
  long second;
};
struct SecondNested {
  long first;
  long second;
};
struct Outer {
  FirstNested first;
  SecondNested second;
};
```

Unions

- Have the type discriminator field:

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

Unions (contd.)

- If the **`default`** option exists, it must be possible to choose it:

```
union U switch (boolean) {
  case FALSE:
    long count;
  case TRUE:
    string message;
  default: //illegal, cannot happen
    float cost;
};
```

Unions (contd.)

- A special case: an optional value

```
union AgeOpt switch (boolean) {
  case TRUE:
    unsigned short age;
};
```

Unions (contd.)

- Simulation of function overloading (not recommended):

```
enum InfoKind { text, numeric, none };
union Info switch (InfoKind) {
  case text:
    string description;
  case numeric:
    long index;
};

interface Order {
  void set_details(in Info details);
};
```

Unions (contd.)

- The recommended solution:

```
interface Order {
  void set_text_details(in string details);
  void set_details_index(in long details);
  void clear_details();
};
```

Arrays

- Single- and multidimensional:

```
typedef Color ColorVector[10];
typedef string IDtable[10][20];
```

- **`typedef`** obligatory:

```
Color ColorVector[10]; //invalid
```

- All dimension obligatory:

```
typedef string IDtable[][20]; //invalid
```

- Passing indices between client and server not portable - initial index depends on the programming language

Sequences

- Variable-length vectors, with the possibility to establish the maximum length:

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

- Sequences of sequences are possible:

```
typedef sequence<Numbers> ListOfNumberVectors;
```

- The element type can be anonymous (not recommendes because of problems with initialization):

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

Sequences and Arrays

- What to use when:
  - Variable length list - sequence
  - Constant length list - array
  - Recursive data structures - sequence
  - Sparse matrix - sequence
- Sparse matrix:

```
typedef long Matrix[100][100];
interface MatrixProcessor {
  Matrix invert_matrix(in Matrix m);
};
```

Sequences and Arrays (contd.)

```
struct NonZeroElement {
  unsigned short row;
  unsigned short col;
  long val;
};
typedef sequence<NonZeroElement> Matrix;

interface MatrixProcessor {
  Matrix invert_matrix(in Matrix m);
};
```

Recursive Types

- Recursion via structures:

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

Recursive Types (contd.)

- Recursion via unions:

```
enum OpType {OP_AND, OP_OR, OP_NOT, OP_BITAND, OP_BITOR,
             OP_BITXOR, OP_BITNOT};
enum NodeKind {LEAF_NODE, UNARY_NODE, BINARY_NODE};
union Node switch (NodeKind) {
case LEAF_NODE:
  long value;
case UNARY_NODE:
  struct UnaryOp {
    OpType            op;
    sequence<Node, 1> child;
  } u_op;
case BINARY_NODE:
  struct UnaryOp {
    OpType            op;
    sequence<Node, 2> children;
  } bin_op;
};
```

Recursive Types (contd.)

- Recursion can be expressed by sequences only:

```
//...
case BINARY_NODE:
  struct UnaryOp {
    OpType op;
    Node   children[2]; // Illegal recursion, not a sequence
  } bin_op;
//...
```

Forward Declarations

- Allow to avoid anonymous types:

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

Forward Declarations (contd.)

- Forward declarations can refer to structures and unions
- Until there is no definition, the type is incomplete
- Incomplete types can appear only in declaration of sequences
- The sequence of incomplete types is an incomplete type

Forward Declarations (contd.)

```
struct Foo; // Introduces Foo type name,
// Foo is incomplete now
// ...
struct Foo {
// ...
}; // Foo is complete at this point
```

- In the recursive structures and unions the incomplete member sequences can refer only to the incomplete types just being defined

```
struct Foo; // Forward declaration
typedef sequence<Foo> FooSeq;
struct Bar {
  long value;
  FooSeq chain; //Illegal, Foo is not an enclosing
                //struct or union
};
```

Mutually Recursive Structures

```
// Not legal IDL!
typedef something Adata;
typedef whatever  Bdata;
struct Astruct {
  Adata                data;
  sequence<Bstruct, 1> nested; //illegal
};
struct Bstruct {
  Bdata                data;
  sequence<Astruct, 1> nested;
};
```

Mutually Recursive Structures (contd.)

```
typedef something Adata;
typedef whatever  Bdata;
enum StructType { A_TYPE, B_TYPE };
union ABunion switch (StructType) {
case A_TYPE:
  struct Acontents {
    Adata               data;
    sequence<ABunion,1> nested;
  } A_member;
 struct Bcontents {
    Bdata               data;
    sequence<ABunion,1> nested;
  } B_member;
};
```

Constants and Literals

- Only for basic types, not for compound types:

```
const float   PI = 3.1415926;
const char    NUL = ’\0’;
const string  LAST_WORDS = ”My god, it’s full of stars”;
const octet   MSB_MASK = 0x80;

enum Color    { red, green, blue };
const Color   FAVOURITE_COLOR = green;

const boolean CONTRADICTION = FALSE; //bad idea
const long    ZERO = 0; //bad idea
```

Constants and Literals (contd.)

- Other names (aliases) of the basic types can be used to define constants:

```
typedef short TempType;
const TempType MAX_TEMP = 35;
```

- All C++ literal types available:

```
const string S1= ”Quote: \””;
const string S2= ”hello world”;
const string S3= ”hello” ” world”;
const string S4= ”\xA” ”B”; // two characters
const string<5> B5 = ”Hello”;
```

Constant Expressions

- Operators:
  - arithmetic: `+ - * / %`
  - bitwise: `| & ^ << >> ~`
- Integer- or floating-point expressions, not mixed ones:

```
const short MIN_TEMP = -10;
const short MAX_TEMP = 35;
const short AVG_TEMP = (MAX_TEMP + MIN_TEMP) / 2;

const float TWICE_PI = 3.14 * 2.0; // Can't use
                                   // 3.14 * 2 here
```

Constant Expressions (contd.)

- Bitwise operations:
  - Shifting of (unsigned) short by more than 16 bits or (unsigned) long by more than 32 bits has an undefined effect
  - >> is a logical shift

```
const long ALL_ONES = -1; //0xffffffff
const long LHW_MASK = ALL_ONES << 16; //0xffff0000
const long RHW_MASK = ALL_ONES >> 16; //0x0000ffff
```

Interfaces and Operations

```
interface Haystack {exception NotFound {
        unsigned long num_straws_searched;
    };

    const unsigned long MAX_LENGHT = 10;      //Max len of a needle

    readonly attribute unsigned long num_straws;   //Stack size

    typedef long Needle;    //ID type for needles
    typedef string straw;  //ID type for straws

    void    add (in Straw s);        //Grow Stack
    boolean remove(in Straw s);          //Shrink Stack
    void   find(in Needle n) raises(NotFound)   //Find Needle
};
```

Interfaces and Operations (contd.)

- Scope rules the same as in C++

```
interface FeedShed {
    typedef sequence<Haystack> Stacklist;

    Stacklist feed_on_hand();//Return all stacks in shed

    void add(in Haystack s); //Add another haystack
    void eat(in Haystack s); //Cows need to be fed

    //look for a needle in all haystacks
    boolean find(in Haystack::Needle n)
               raises(Haystack::Notfound);

    //Hide a needle (note that this is a oneway operation)
    oneway void hide(in Haystack s, in Haystack::Needle n);
};
```

- Interface names are types
- Interfaces can be passed as parameters

Interface Communication Model

- How the needles get from the feedshed to the haystack?
  - IDL does not explain that, we can only guess
  - Maybe they drop from the farmer's pocket?
- IDL operations and attributes are the only sources defining interactions between objects
- If it is not defined in IDL, for CORBA it does not exist
- There is some hidden communication between **`Haystack`** and **`FeedShed`** objects
- It can make later modifications of the system (separation of objects) more difficult

Directional Attributes

- The use of directional attributes
  - Improves efficiency
  - Establishes the memory-management responsibility
- Definition style
  - The return value is emphasized

```
interface Primes {
  typedef unsigned long prime;
  prime  next_prime(in long n);
  void   next_prime2(in long n, out prime p);
  void next_prime3(inout long n);
};
```

Illegal Constructs

- Overloading
  - `prime  next_prime(in long n);`
  - `void   next_prime(in long n, out prime p);`
  - `void next_prime(inout long n);`
- Anonymous types
  - `sequence<long> get_longs();`
  - `void get_octets(out sequence<octet> s);`
- `const` operations
  - `SomeType read_value() const;`

User-defined Exceptions

```
exception Failed {};
exception RangeError {
  unsigned long supplied_val;
  unsigned long min_permitted_val;
  unsigned long max_permitted_val;
};
```

- Exceptions cannot be nested and cannot be components of other types

```
struct ErrorReport
{
  Object obj;
  RangeError exc; // Error
};
```

- Operation use the keyword raises to indicate possible exceptions thrown

```
interface Unreliable {
  void can_fail() raises(Failed);
  void can_also_fail() raises(Failed, RangeError);
};
```

- Lack of exception inheritance

Exception Design

- Exceptions only in exceptional situations
- Containing useful information
- Containing exact information
- Containing complete information
- API convenient for user, not for the implementor
- Do not use return values as the error indicators

System Exceptions

- Every operation can throw a system exception
- It is not specified explicitly

```
void op1() raises(BAD_PARAM);    //BAD
```

- There are 39 system exceptions
- System exceptions have two fields:
  - **`completion_status completed;`**
    - “Yes”, “No”, “Maybe”
  - **`unsigned long minor;`**
    - Additional information about the error

oneway Operations

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

- Unreliable data transfer in one direction
  - return type **`void`**
  - no **`out`** nor **`inout`** parameters
  - no exception specification
- „Best effort” and „at most once” semantics
- No guarantee of non-blocking behaviour and message ordering
- CORBA Messaging gives better control over such calls

Attributes

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

- Equivalent to:

```
interface Thermostat {
  short get_temperature();
  short get_nominal_temp();
  void set_nominal_temp(in short t);
};
```

- User-defined exceptions not possible

Modules

- Similar to namespaces in C++
- Prevent the pollution of the global namespace
- Can be nested, closed and reopened

```
module A {
    typedef short number;
    typedef string name;
};
module B {
    interface C {
        A::number age();
        A::name first_name();
        A::name last_name();
    };
};
module A { // modules can be re-opened
    interface C { // doesn’t interfere with B::C
        number age(); // just need ‘number’, not ‘A::number’
    };
};
```

Forward Declarations

- Used with mutually-dependent interfaces:

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

- You cannot declare an interface in another module in this way:

```
module Females {
  interface Males::Husband;
//error
};
```

- It can be done in the following way:

```
module Females {
  interface Wife;
};
module Males {
  interface Husband {
    Females::Wife get_spouse();
  };
};
module Females {
  interface Wife {
    Males::Husband
get_spouse();
  };
};
```

Inheritance

```
interface Thermometer {
  typedef short TempType;
  readonly attribure TempType temperature;
};
interface Thermostat : Thermometer {
  void set_nominal_temp(in TempType t);
};
```

- Polymorphism and scope rules as in C++
- Implicit inheritance from the `Object` interface
- Inheritance of interfaces, not implementations

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

Inheritance - An Empty Interface

```
interface Vehicle {};
interface Car: Vehicle {
  void start();
  void stop();
};
interface Airplane: Vehicle {
  void take_off();
  void land();
};
interface Garage {
  void park(in Vehicle v);
  void make_ready(in Vehicle v);
};
```

Inheritance - Redefinitions

- Types, constants and exceptions can be redefined in the derived interface:

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

Inheritance - Redefinitions (contd.)

- Attributes nor operations cannot be redefined in the derived interface:

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

Inheritance - Redefinitions (contd.)

- Attribute and operation overloading is also illegal:

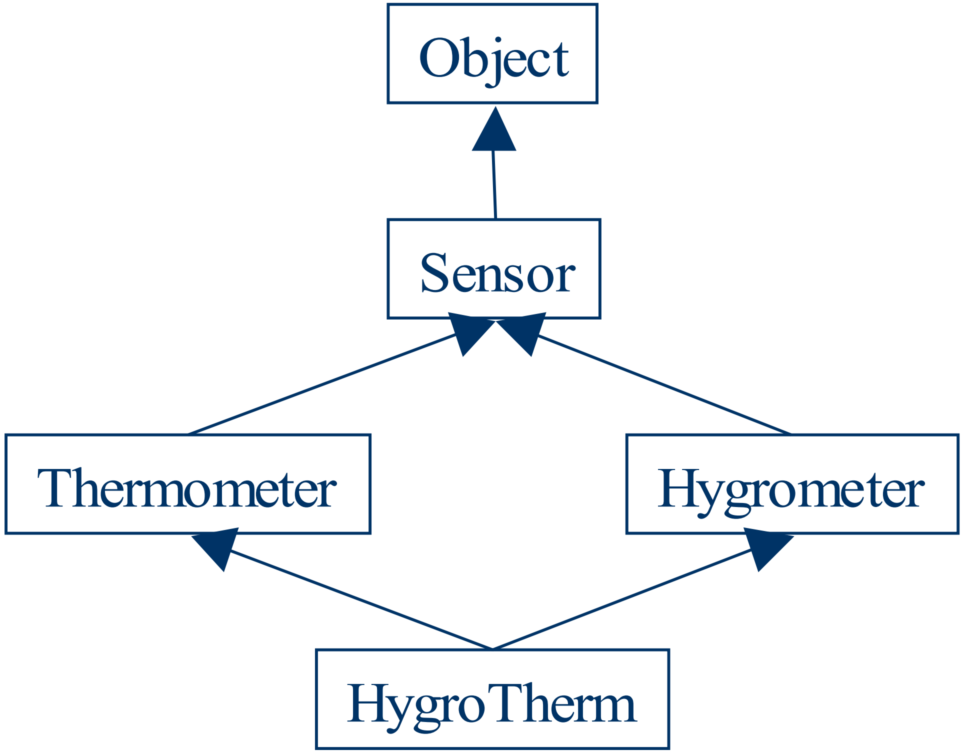
```
interface Thermometer {
  attribute string my_id;;
  string get_id();
  void set_id(in string s);
};
interface Thermostat : Thermometer {
  attribute double my_id; //error
  double get_id(); //error
  void set_id(in double d); //error
};
```

Multiple Inheritance

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

- We can derive from more than one base interface:

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



Multiple Inheritance Limitations

- Operations and attributes can be inherited only from one base interface:

```
interface Thermometer {
  attribute string model;
  void initialize();
};
interface Hygrometer {
  attribute string model;
  void initialize();
};
interface HygroTherm : Thermometer, Hygrometer {
 // ... ambiguity - which model and initialize()
};
```

Multiple Inheritance Limitations (contd.)

- Conflicting type definitions:

```
interface Thermometer {
typedef string<16> ModelType;
};
interface Hygrometer {
  typedef string<32> ModelType;
};
interface HygroTherm : Thermometer, Hygrometer {
attribute ModelType model; //ambiguity - 16 or 32 chars?
};
```

- Explicit scope specification:

```
interface HygroTherm : Thermometer, Hygrometer {
  attribute Thermometer::ModelType model; // OK, 16 chars
};
```

Names and Scope

- Identifiers unique within their scope
  - module, interface, structure, union, exception, operation definition
- Lower- and uppercase letters significant in identifiers, but two identifiers differing only in case are not allowed

```
module CCS {
  typedef short TempType;
  typedef double temptype; //Error
};
```

- Name in the scope cannot be the same, as the name of the surrounding scope - the following is illegal:

```
module A {
    module A {
         /* . . . */
    };
};
```

```
interface SomeName {
  typedef long SomeName;
};
```

- Names found by searching of subsequent surrounding scopes - as in C++

IDL Repository ID

```
module CCS {
  typedef short TempType;
  interface Thermometer {
    readonly attribute TempType temperature;
  };
  interface Thermostat : Thermometer {
    void set_nominal_temp(in TempType t);
  };
};
```

```
IDL:CCS:1.0
IDL:CCS/TempType:1.0
IDL:CCS/Thermometer:1.0
IDL:CCS/Thermometer/temperature:1.0
IDL:CCS/Thermostat:1.0
IDL:CCS/Thermostat/set_nominal_temp:1.0
```

#pragma prefix

```
#pragma prefix ”acme.com”
module CCS {
  // ...
};

IDL:acme.com:/CCS:1.0
IDL:acme.com:/CCS/Temptype:1.0
IDL:acme.com:/CCS/Thermometer:1.0
IDL:acme.com:/CCS/Thermometer/temperature:1.0
IDL:acme.com:/CCS/Thermostat:1.0
IDL:acme.com:/CCS/Thermostat/set_nominal_temp:1.0
```

- Prefix reduces the risk of name collision
- Prefix does not influence the generated code

Recent Additions to IDL

- Wide characters and strings (unicode)
  - `const wchar C =L’X’;`
  - `const wstring GREETING = L"Hello"`
- 64-bit integers
  - `long long`
  - `unsigned long long`
- Extended floating-point type
  - `long double`
  - mantissa ≥64 bits, exponent ≥15 bits
- Fixed-point decimal numbers
  - `typedef fixed<9,2> AssetValue;`
  - `const fixed val1=3.14D;`