# Java IDL (CORBA)

- introduced in Version 1.2 of the Java 2 platform,
  - provides an interface between Java programs and distributed objects and services built using the Common Object Request Broker Architecture (CORBA).
- CORBA is defined by the Object Management Group (OMG).
  - describes an architecture, interfaces, and protocols that distributed objects can use to interact with each other.
  - Interface Definition Language (IDL) is an implementation-independent language for describing the interfaces of remote-capable objects.
- Standard mappings for converting IDL interfaces
  - into C++ classes, C code, and Java classes,
  - generated classes use the underlying CORBA framework to communicate with remote clients
  - Java IDL is Sun's implementation of the standard IDL-to-Java mapping
  - standard Java SDK in the org.omg.CORBA package, the org.omg.CosNaming package, and other org.omg .* packages.

# CORBA vs. Java RMI

- Like RMI, Java IDL gives you a way to access remote objects over the network.
- It also provides the tools you need to make your objects accessible to other CORBA clients.
- If you export a Java class using Java IDL, it's possible to create an instance of that class and publish it through a naming/directory service.
- A remote client can find this object, call methods on it, and receive data from it, just as if it were running on the client's local machine.
- Unlike RMI, however, objects that are exported using CORBA can be accessed by clients implemented in any language with an IDL binding (not just Java).

# A Note on Evolving Standards

- At the time of Java 2 Version 1.2, the CORBA specification and the IDL-to-Java binding for CORBA were in a bit of flux.
  - The server-side object adaptor interface had been altered significantly by the OMG in Version 2.3 of the CORBA specification.
  - The Basic Object Adaptor (BOA) interface had been replaced by the Portable Object Adaptor (POA).
  - This filled a gap in the specification left by the BOA that led to vendor-specific extensions and, therefore, CORBA server objects that were dependent on particular vendor ORB implementations.
- IDL-to-Java mapping took some time to be updated to support POA
  - JDK 1.2 was released before the new version of the Java mapping.
  - By the time JDK 1.4 was introduced in beta in 2001, the POA-compatible version of the IDL-to-Java mapping had been released, and the Java IDL packages, as well as the IDL-to-Java compiler in JDK 1.4, were based on this mapping.

# Standards (contd.)

- Interoperable Naming Service (INS) interface adds new utilities and functionality on top of the standard CORBA Naming Service.
  - INS was incorporated into the CORBA 2.3 specification, and support for it in Java IDL was introduced in JDK 1.4.
  - If you are using JDK 1.4 or later, you are using a POA-compatible mapping of the CORBA interfaces.
  - If you are using JDK 1.3 or JDK 1.2, you are using the "pre- POA" version of the IDL-to-Java mapping that Sun used prior to adding the POA support.
  - JDK 1.4 or later has access to the INS interface and the Naming Service provided with the Java IDL.
  - The Object Request Broker (ORB) supports these extended features.

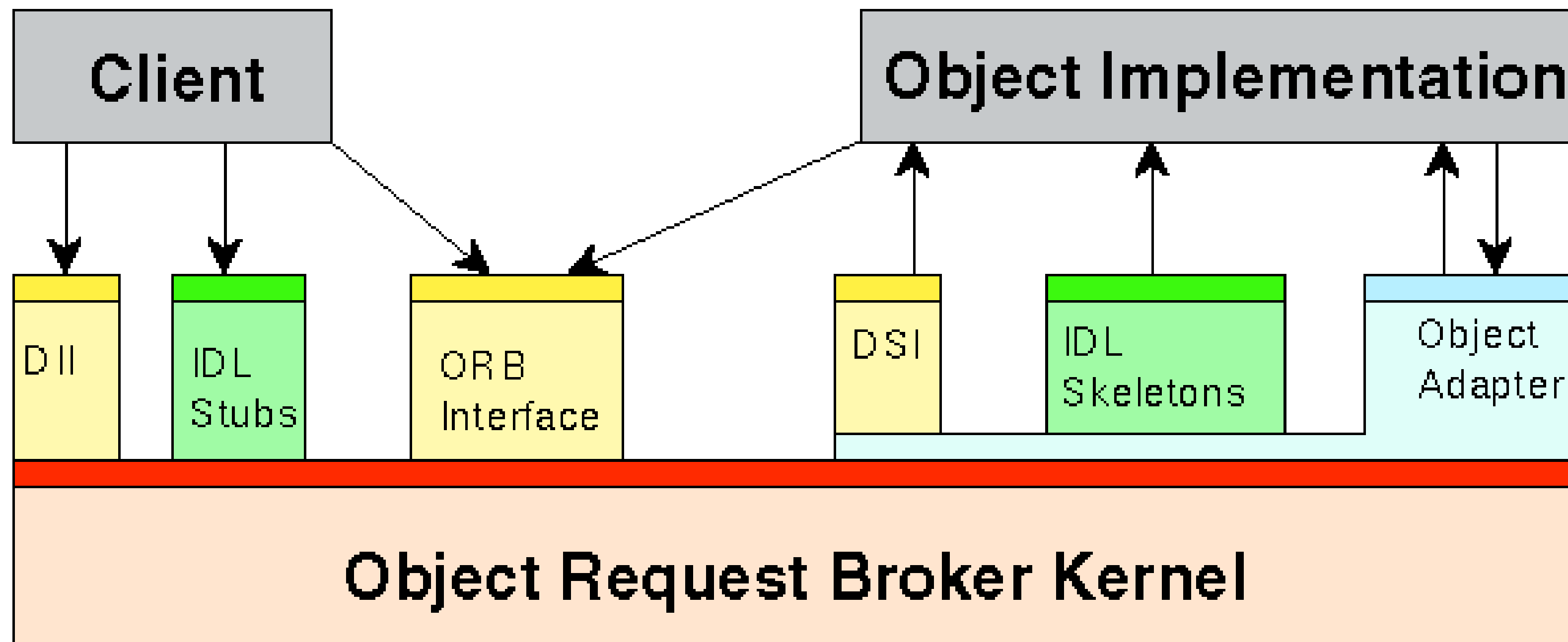
# CORBA Architecture

- CORBA was designed from the start to be a language-independent distributed object standard,
  - so it is much more extensive and detailed in its specification than RMI is (or needs to be).
  - Extra details are required in CORBA because it needs to support languages that have different built-in features.
  - Some languages, like C++, directly support objects, while others, like C, don't.
- The CORBA specifies an object model so that non object-oriented languages can take advantage of CORBA.
  - Java includes built-in support for communicating object interfaces and examining them abstractly (using Java bytecodes and the Java Reflection API).
  - CORBA specification includes details about a Dynamic Invocation Interface and a Dynamic Skeleton Interface, which can be implemented in languages that don't have their own facilities for these operations.
  - Generally, there needs to be a mapping between the built-in features and the features as defined by the CORBA specification.

# Interface Definition Language

- The Interface Definition Language provides the primary way of describing data types in CORBA.
  - IDL is independent of any particular programming language.
  - Mappings, or bindings, from IDL to specific programming languages are defined and standardized as part of the CORBA specification.
  - Standard bindings for C, C++ , Smalltalk, Ada, COBOL, Lisp, Python and Java have been approved by the OMG.
- The central CORBA functions, services, and facilities, such as the ORB and the Naming Service, are also specified in IDL.
  - This means that a particular language binding also specifies the bindings for the core CORBA functions to that language.
  - Sun's Java IDL API follows the Java IDL mapping defined by the OMG standards.

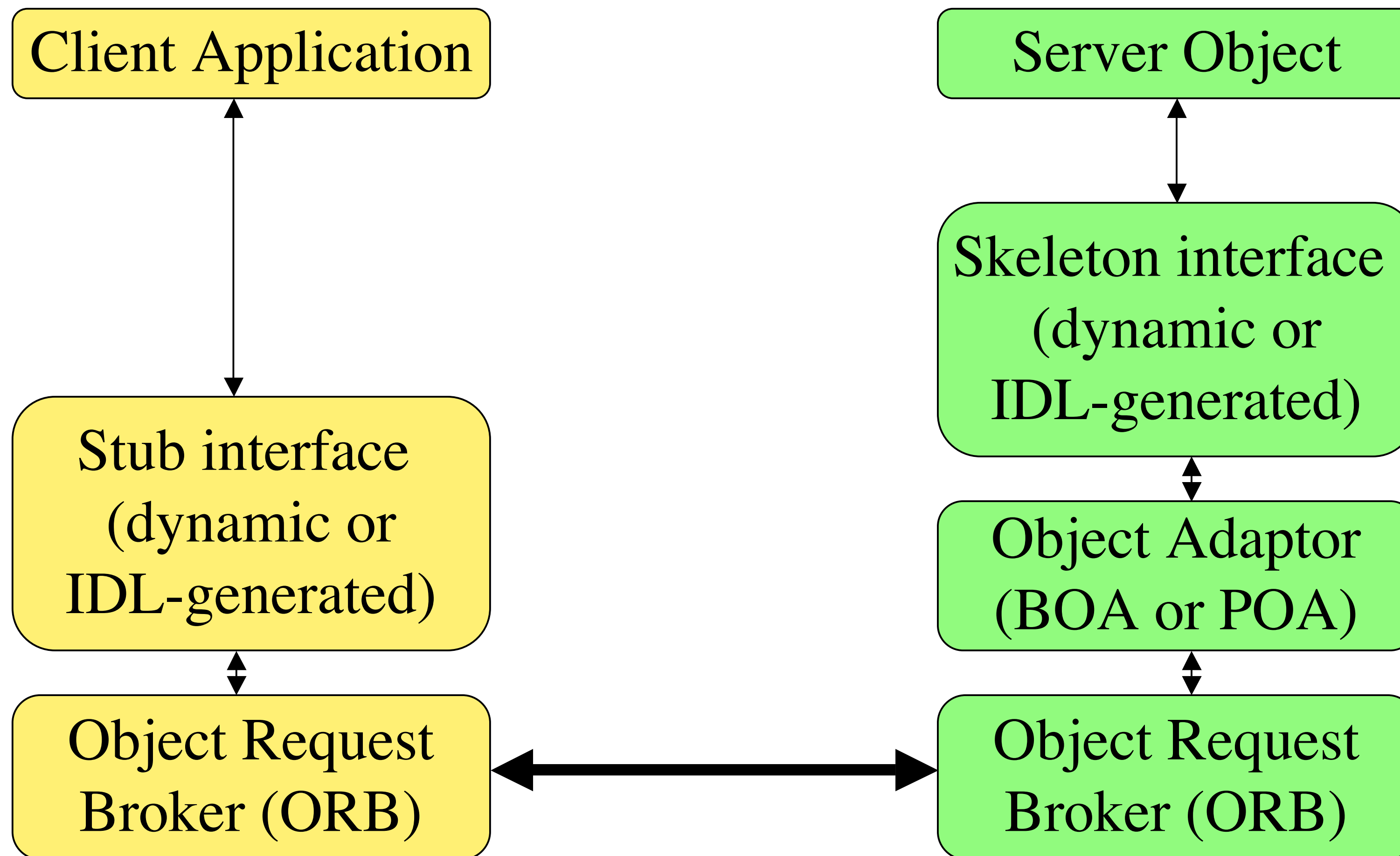
# Basic CORBA Architecture



# The Object Request Broker and the Object Adaptor

- The core of CORBA is the Object Request Broker (ORB)
  - Each machine involved in a CORBA application must have an ORB running in order for processes on that machine to interact with CORBA objects running in remote processes.
  - Object clients and servers make requests through their ORBs
  - the ORB is responsible for making the requests happen or indicating why they can't.
- The client ORB provides a stub for a remote object.
  - Requests made on the stub are transferred from the client's ORB to the ORB servicing the implementation of the target object.
  - The request is passed on to the implementation through an object adaptor and the object's skeleton interface.

# Remote Object Invocation



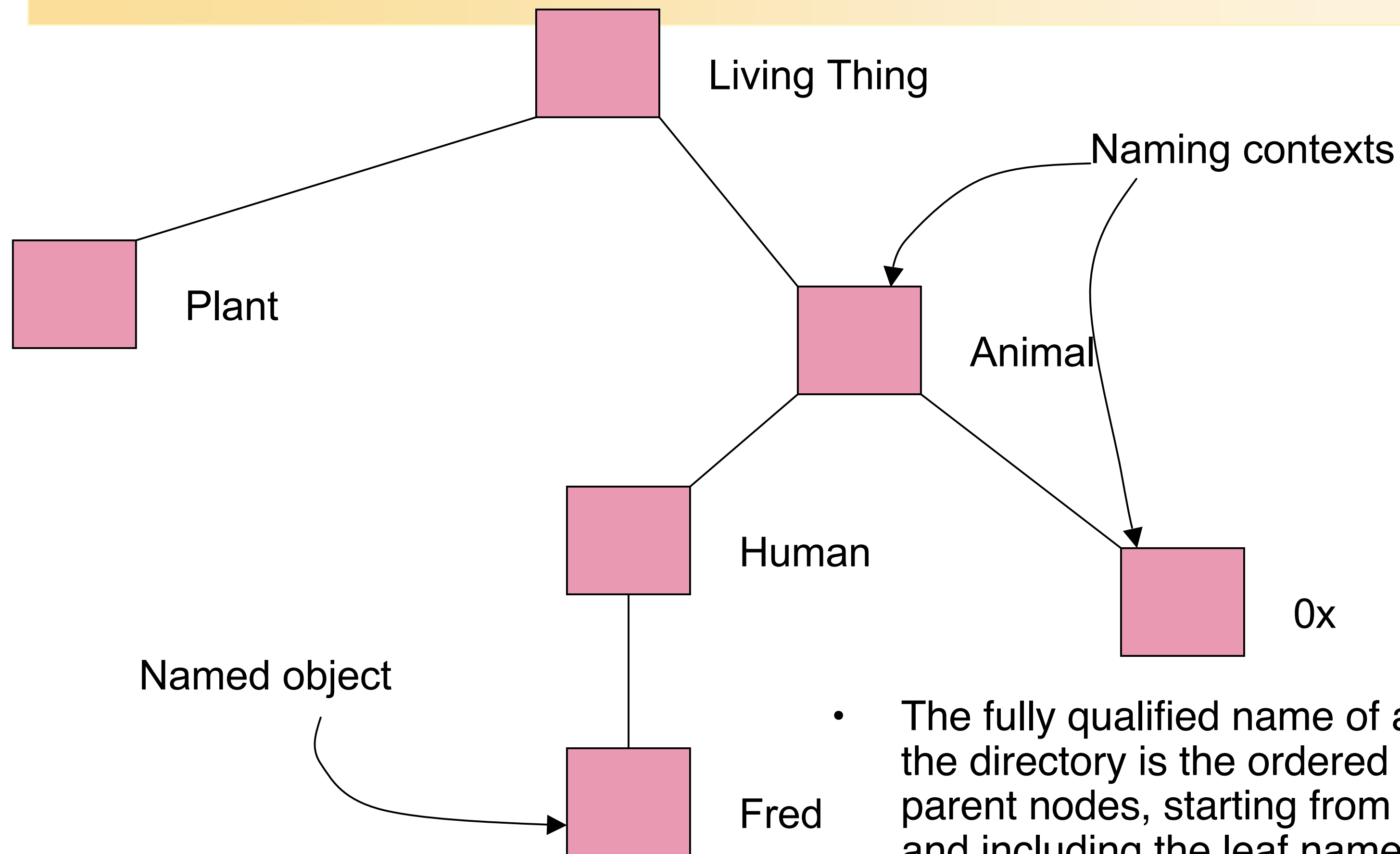
# The Skeleton interface

- The skeleton interface is specific to the type of object that is exported remotely through CORBA.
  - provides a wrapper interface that the ORB and object adaptor can use to invoke methods on behalf of the client or as part of the lifecycle management of the object.
  - The object adaptor provides a general facility that "plugs" a server object into a particular CORBA runtime environment. (BOA vs. POA)
- All server objects can use the object adaptor to interact with the core functionality of the ORB,
  - the ORB can use the object adaptor to pass along client requests and lifecycle notifications to the server object.
  - Typically, an IDL compiler is used to generate the skeleton interface for a particular IDL interface; this generated skeleton interface will include calls to the object adaptor that are supported by the CORBA environment in use.

# The Naming Service

- The CORBA Naming Service provides a directory naming structure for remote objects.
  - The CORBA Naming Service is one of the naming and directory services supported by JNDI,
  - Concepts used in its API are similar to the general model of Contexts and DirContexts used in JNDI.
  - The naming tree always starts with a root node, and subnodes of the object tree can be created by an application.
  - Actual objects are stored by name at the leaves of the tree.

# A Naming Directory




- The fully qualified name of an object in the directory is the ordered list of all of its parent nodes, starting from the root node and including the leaf name of the object itself.

# Inter-ORBA Communication

- CORBA v.2.0 standard includes specifications for inter-ORB communication protocols that can transmit object requests between various ORBs running on the network.
  - independent of the particular ORB implementations running at either end of the communication link.
  - An ORB implemented in Java can talk to another ORB implemented in C, as long as use the same CORBA communication protocol.
  - The inter-ORB protocol is responsible for delivering messages between two cooperating ORBs (method requests, return types, error messages, etc.)
  - The inter-ORB protocol also deals with differences between the two ORB implementations, like machine-level byte ordering and alignment.
- The Internet Inter-ORB Protocol (IIOP) is an inter-ORB protocol based on TCP/IP.
  - TCP /IP is by far the most commonly used network protocol on the Internet, so IIOP is the most commonly used CORBA communication protocol.
  - There are other standard CORBA protocols defined for other network environments, however( e.g.; DCE Common Inter-ORB Protocol - DCE-CIOP),

# Creating CORBA Objects

- In order to distribute a Java object over the network using CORBA, you have to define your own CORBA-enabled interface and its implementation.

This involves doing the following:

- Writing an interface in the CORBA Interface Definition Language
- Generating a Java base interface, plus a Java stub and skeleton class, using an IDL-to-Java compiler
- Writing a server-side implementation of the Java base interface

# IDL Primer

- The syntax of both Java and IDL were modeled on C++
  - Interfaces in IDL are declared much like classes in C++ and, thus, classes or interfaces in Java.
- The major differences between IDL and Java are:
- IDL is a declaration language.
  - In IDL, you declare only the names and types for interfaces, data members, methods, method parameters, etc.
  - Method implementations are created in the implementation language you choose (in this case Java), after you've used an IDL compiler to convert your IDL interface to your target language.I
- DL, like C++ , includes nonclass data structure definitions, like structs, unions, and enumerations.

# IDL Primer (contd.)

- Method parameters in IDL include modifiers that specify whether they are input, output, or input/output variables.
  - In Java, all primitive data types are passed by value, and all object data types are passed by reference.
- An IDL file can include multiple public interfaces.
  - Only a single public class can be defined in a given Java file (although Java does allow for multiple inner classes within a single public class definition, and multiple nonpublicclasses per file).
  - Modules, which are similar to Java packages, can be nested within other modules in the same IDL file, and interfaces in multiple distinct modules can bedefined in the same IDL file.
  - In Java, you can define a class only within a single package in a single Java file.

# Modules

- Modules are declared in IDL using the module keyword,
  - followed by a name for the module and an opening brace that starts the module scope.
  - Everything defined within the scope of this module (interfaces, constants, other modules) falls within the module and is referenced in other IDL modules using the syntax modulename::x.

```
// IDLmodule jent
   {
      module corba {interface NeatExample ...}
        ;
   } ;
```

  - If you want to reference the NeatExample interface in other IDL files, you use the syntax jent::corba::NeatExample

# Interfaces

- Interfaces declared in IDL are mapped into classes or interfaces in Java.
  - IDL is used only to declare modules, interfaces, and their methods.
  - Methods on IDL interfaces are always left abstract, to be defined in the programming language you use to implement the interfaces.
- The declaration of an interface includes an interface header and an interface body.
  - The header specifies the name of the interface and the interfaces it inherits from (if any).

```
interface PrintServer : Server { ...
```

- This header starts the declaration of an interface called PrintServer
  - inherits all the methods and data members defined in the Server interface.
  - An IDL interface can inherit from multiple interfaces; simply separate the interface names with commas in the inheritance part of the header .

# Data Members and Methods

- The interface body declares all the data members (or attributes) and methods of an interface.
  - Data members are declared using the attribute keyword.
  - At a minimum, the declaration includes a name and a type
  - The declaration can optionally specify whether the attribute is read-only or not, using the readonly keyword.
  - By default, every attribute you declare is readable and writable (for Java, this means that the IDL compiler generates public read and write methods for it).

```
readonly attribute string myString;
```

  - You declare a method by specifying its name, return type, and parameters
  - You can also optionally declare exceptions the method might raise, the invocation semantics of the method, and the context for the method call

```
string parseString(in string buffer);
```

  - This declares a method called parseString() that accepts a single string argument and returns a string value.

# A Complete IDL Example

```
module OS {
  module services {
      interface Server {
            readonly attribute string serverName;
            boolean init(in string sName);
       };
       interface Printable {
            boolean print(in string header);
       };
       interface PrintServer : Server {
            boolean printThis(in Printable p);
};};};
```

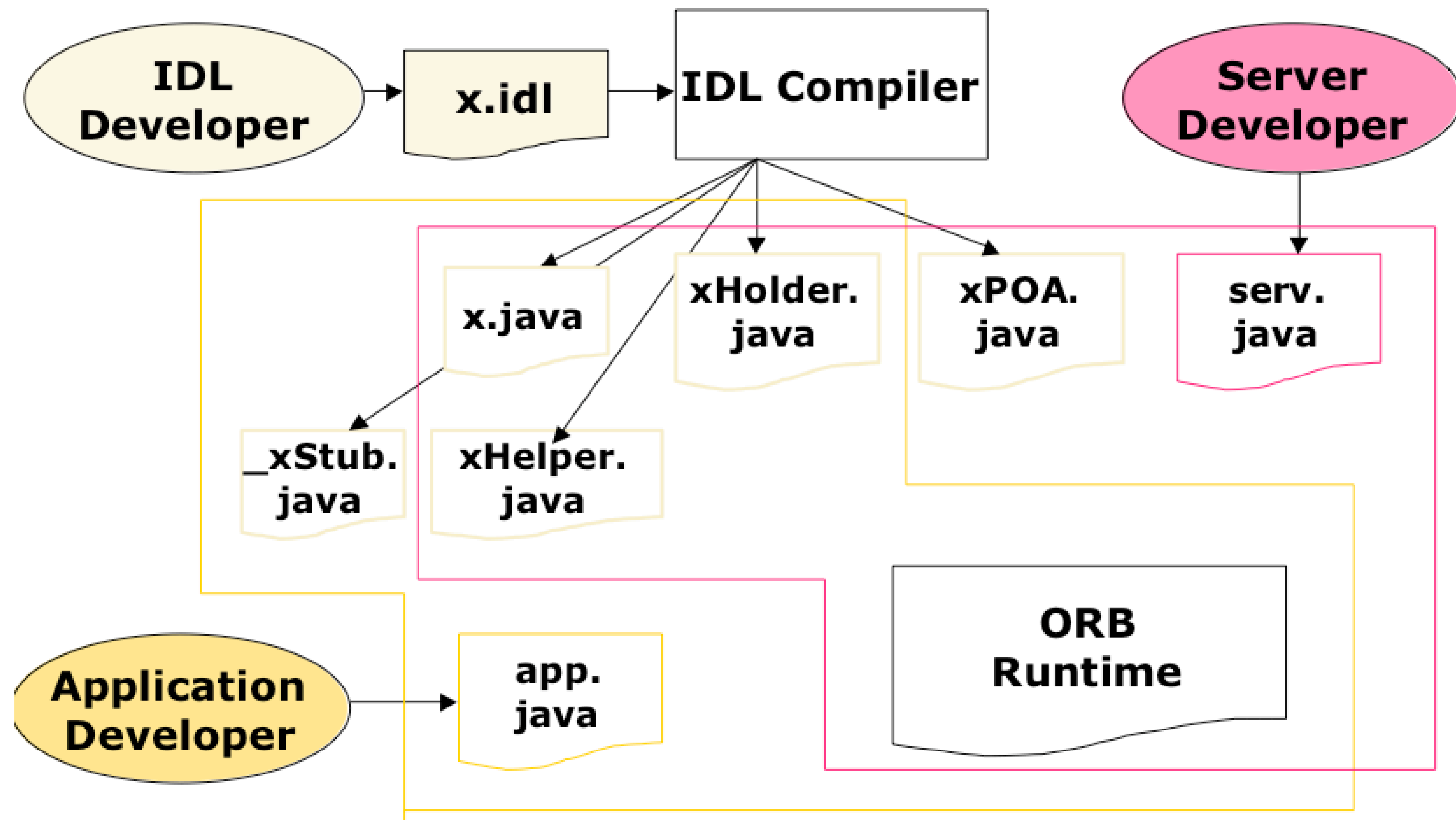
# Turning IDL into Java

These classes comprise the "outward-facing" mapping of the CORBA object's interface:

- (the interfaces that clients of the object use directly).

- A java interface with the same name as the IDL interface (e.g., Server).
- A helper class whose name is the name of the IDL interface with "Helper" appended to it (e.g., ServerHelper).
- A holder class whose name is the name of the IDL interface with "Holder" appended to it (e.g., ServerHolder).
- A client stub class, called _interface-nameStub, that acts as a client-side implementation of the interface.

# IDL-to-Java translation



# IDL-to-Java (server-side)

- The IDL-to-Java compiler can also generate server-side skeleton classes
  - Can be used for the server-side implementation of the remote CORBA interface.
- Pre-POA:
  - server skeleton class called _interface-nameImplBase (e.g., _ServerImplBase), which is a base class for a server-side implementation of the interface.
- POA:
  - server skeleton class named interfaceNamePOA (e.g, ServerPOA), which implements a generated interfaceNameoperations interface and extends the POA-related server-side interfaces.
- Inheritance-based approach

# Delegation-based Server Side implementation

- So far, server-side implementation depends on directly extending a generated class
  - interfaceNamePOA or _interfaceNameImplBase
- The delegation model is based on a scheme in which a server-side delegate is generated by the IDL compiler.
  - This delegate extends the generated skeleton class, and implements each of the mapped remote methods by delegating the incoming method request to a delegate object.
  - This delegate object needs to implement the interfaceNameOperations interface generated by the IDL compiler, but it doesn't have to extend a concrete or abstract base class.
  - This can prove to be useful in cases where you have a preexisting Java class with its own inheritance scheme and want to "export" this class through CORBA for remote access.

# A simple Server class

```
module oreilly {
  module jent {
    module corba {
      // Forward-declare the Account interface,
      interface Account;
      // typedefs: a list of Accounts and a list of floats
      typedef sequence<Account> AccountList;
      typedef sequence<float> floatList;

      exception InsufficientFundsException {};
      interface Account {
         string getName();
        float getBalance();
        void withdraw(in float amt)
                         raises (InsufficientFundsException);
        void deposit(in float amt);
         void transfer(in float amt, in Account src)
                         raises (InsufficientFundsException);
         void transferBatch(in floatList amts, in AccountList srcs)
                         raises (InsufficientFundsException);
};};};};
```

# Generating Java classes

- Run the idl compiler:
  - C:\>idlj -fall Account.idl
- This creates five Java classes:
  - a Java version of the interface,
  - a helper class,
  - a holder class,
  - a client stub, and
  - a server skeleton.
- The -fall option tells the compiler to generate both client-side and server-side mapping interfaces.

# Helper class - AccountHelper

- The helper class is a standalone utility class that doesn't extend any other interfaces:

```
abstract public class AccountHelper {
```

- static methods that let you read and write Account objects to and from CORBA I/O streams:

```
public static oreilly.jent.corba.Account read
(org.omg.CORBA.portable.lnputStream istream)
```

```
public static void write (org.omg.CORBA.portable.autputStream
ostream, oreilly.jent.corba.Account value)
```

- a type () method that provides the TypeCode for the mapped Account class:

```
synchronized public static org.omg.CORBA.TypeCode type ()
```

- a narrow( ) method that safely narrows a CORBA org.omg.CORBA object reference into an Account reference:

```
public static oreilly.jent.corba.Account narrow
(org.omg.CORBA.object obj)
```

- Object narrowing is CORBA's equivalent to directly casting object references

# Holder class

- A holder class for the Account class, implements the CORBA Streamable interface:
- public final class AccountHolder implements `org.omg.CORBA.portable.Streamable`
- The holder class is a wrapper used when Account objects are called for as out or inout arguments in an IDL method.
- All holder classes implement the Streamable interface from the org.omg.CORBA.portable package,
  - which includes implementations of the _read ( ) and _write ( ) methods of the Streamable interface:
  - `public void _read (org.omg.CORBA.portable.inputStream i)`
  - `public void _write (org.omg.CORBA.portable.outputStream o)`

# Client Stub

```
// Get the name of the account owner
public String getName (){  org.omg.CORBA.portable.InputStream $in = null;
   try {
     org.omg.CORBA.portable.autputStream $out =
         _request ("getName" , true);
     $in = _invoke ($out) ;
     String $result = $in.read-string();
     return $result;
   } catch (org.omg.CORBA.portable.ApplicationException $ex) {
     $in = $ex.getlnputStream (};
     String _id = $ex.getId ( );
     throw new org.omg.CORBA.MARSHAL (_id);
   } catch (org.omg.CORBA.portable.RemarshalException $rm) {
     return getName ();
   } finally {_
     releaseReply ($in);
}} // getName
```