

Interoperability

John Mitchell

Thanks to Kathleen Fisher for contributions to slides

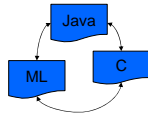
Outline

- ◆ Basic concepts
- ◆ Java Native Interface (JNI)
- ◆ Interface definition languages (IDL)
- ◆ Interoperability by binary compatibility: COM
- ◆ Interoperability by conversion: Corba
- ◆ Interoperability by common platform: .NET

Why is interoperability important?

- ◆ Write each part of a complex system in a language suited to the task:

- C for low-level machine management
- Java/C#/Objective-C for user-interface
- Ocaml/ML for tree transformations



- ◆ Integrate existing systems:

- implemented in different languages
- for different operating systems
- on different underlying hardware systems



What's involved?

- ◆ Languages make different choices:

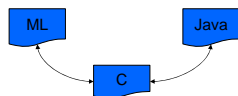
- Function calling conventions
 - caller vs callee saved registers
- Data representations
 - strings, object layout
- Memory management
 - tagging scheme

- ◆ Solution concepts

- Stubs and wrappers
- Data conversion
- "Abstract" treatment of objects
 - Method calls go back to language where object was defined

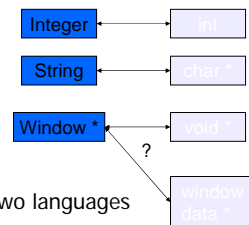
C/C++ as Lingua Franca

- ◆ Ubiquitous
- ◆ Computation model *is* underlying machine:
 - Other languages already understand
 - No garbage collection
- ◆ Representations well-known and fixed
 - Millions of lines of code would break if changed



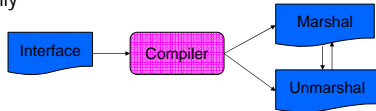
Marshaling and Unmarshaling

- ◆ Convert data representations from one language to another
- ◆ Easier when one end is C as rep is known
- ◆ Policy choice: copy or leave abstract?
- ◆ Tedious, low-level
- ◆ Modulo policy, fixed by two languages



Interface specifications

- ◆ Contract describing what an implementation written in one language will provide for another
 - Inferred from high-level language: JNI
 - Inferred from C header files: SWIG
 - Specified in Interface Definition Language: ocamlidl, COM, CORBA
- ◆ Allow tools to generate marshalling/unmarshalling code automatically



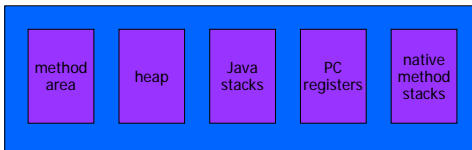
JNI: Integrating C/C++ and Java

- ◆ Java Native Interface
 - Allows Java methods to be implemented in C/C++
 - Such methods can
 - create, inspect, and send messages to Java objects
 - modify Java objects & have changes reflected to system
 - catch and throw exceptions in C that Java will handle
- ◆ JNI enforces policy: pointers are abstract

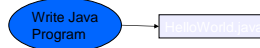
<http://java.sun.com/docs/books/jni/html/start.html>

Recall JVM memory areas

- ◆ Separate memory area for native methods
 - Pass object to native method
 - Convert if primitive type
 - Pass pointer back into Java memory area otherwise



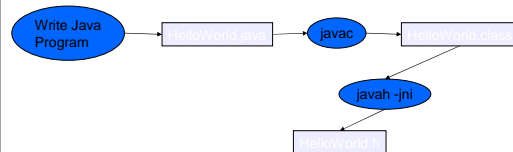
JNI Example: Hello World!

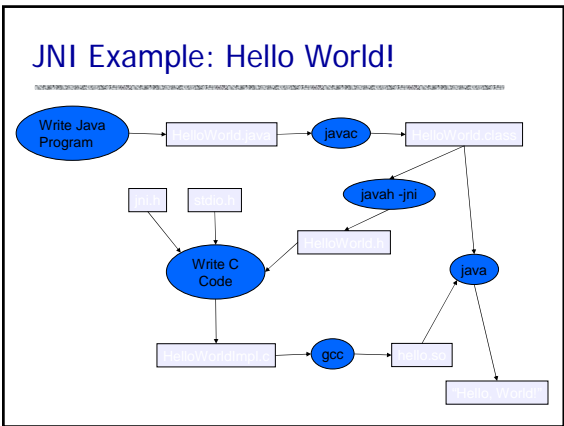
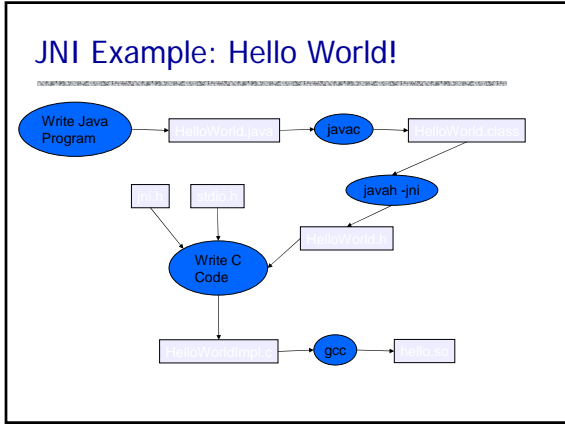
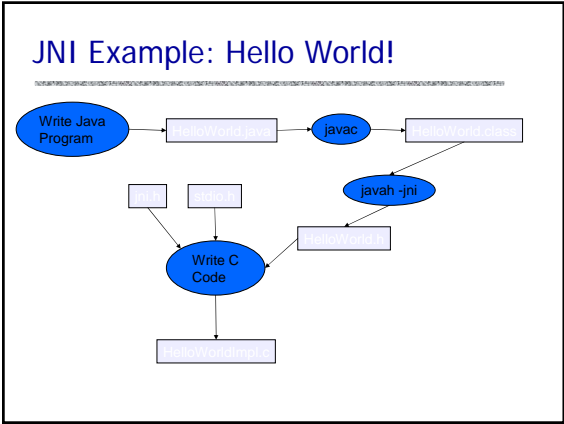


JNI Example: Hello World!



JNI Example: Hello World!





JNI Example: Write Java Code

```

class HelloWorld {
    public native void displayHelloWorld();

    static {
        System.loadLibrary("hello");
    }

    public static void main(String[] args) {
        new HelloWorld().displayHelloWorld();
    }
}

```

JNI Example: Compile Java Code

```

javac HelloWorld.java

```

```

cafe babe 0000 002e 001b 0a00 0700 1207
0013 0a00 0200 120a 0002 0014 0800 130a

```

JNI Example: Generate C Header

```

javah -jni HelloWorld.java

```

```

#include <jni.h>
/* Header for class HelloWorld */
#ifndef _Included_HelloWorld
#define _Included_HelloWorld
#ifdef __cplusplus
extern "C" {
#endif

JNIEXPORT void JNICALL
Java_HelloWorld_displayHelloWorld
(JNIEnv *, jobject);
#endif

```

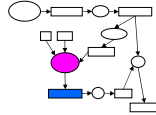
- Function has two "extra" args
 - Environment pointer
 - Provides access in C to JVM functions, e.g., function to convert Java string to char *
 - Object pointer (*this*)

JNI Example: Write C Method

```
#include <jni.h>
#include "HelloWorld.h"
#include <stdio.h>

JNIEXPORT void JNICALL
Java_HelloWorld_displayHelloWorld(JNIEnv *env, jobject obj) {
    printf("Hello world!\n");
    return;
}
```

- Implementation includes 3 header files:
- **jni.h**: provides information that C needs to interact with JVM
 - **HelloWorld.h**: generated in previous step
 - **stdio.h**: provides access to **printf**



JNI Example: Create Shared Lib

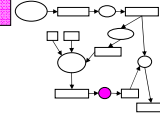
How to create a shared library depends on platform:

Solaris:

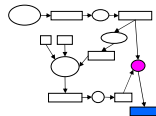
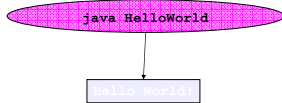
```
cc -g -I/usr/local/java/include \
-I/usr/local/java/include/solaris \
HelloWorldImp.c -o libhello.so
```

Microsoft Windows w/ Visual C++ 4.0:

```
cl -Ic:\java\include \
-Ic:\java\include\win32 \
-LD HelloWorldImp.c -Fehello.dll
```



JNI Example: Run Program



JNI: Type Mapping

- ◆ Java primitive types map to corresponding types in C
- ◆ All Java object types are passed by reference (object)

Native type	Java type	Description
bool	jboolean	unsigned 8 bits
byte	jbyte	signed 8 bits
char	jchar	unsigned 16 bits
short	jshort	signed 16 bits
long	jint	signed 32 bits
long long	jlong	signed 64 bits
float	jfloat	32 bits
double	jdouble	64 bits

JNI: Method Mapping

- The **javah** tool uses type mapping to generate prototypes for native methods:

QuickTime™ and a
TIFF (Uncompressed) decompressor
are needed to see this picture.

JNI: Accessing Java Strings

- ◆ Type **jstring** is **not char ***!
- ◆ Native code must treat **jstring** as an abstract type and use **env** functions to manipulate:

```
JNIEXPORT jstring JNICALL
Java_Prompt_getLine(JNIEnv *env, jobject obj, jstring prompt)
{
    char buf[128];
    const char *str = (*env)-->GetStringUTFChars(env, prompt, 0);
    printf("%s", str);
    (*env)-->ReleaseStringUTFChars(env, prompt, str);
    ...
    scanf("%s", buf);
    return (*env)-->NewStringUTF(env, buf);
}
```

JNI: Calling Methods

- ◆ Native methods can invoke Java methods using the environment argument:

```
JNIEXPORT void JNICALL
Java_Callbacks_nativeMethod(JNIEnv *env, jobject obj, jint depth)
{
    jclass cls = (*env)->GetObjectClass(env, obj);
    jmethodID mid = (*env)->GetMethodID(env, cls, "callback", "(I)V");
    if (mid == 0) {
        return;
    }
    printf("In C, depth = %d, about to enter Java\n", depth);
    (*env)->CallVoidMethod(env, obj, mid, depth);
    printf("In C, depth = %d, back from Java\n", depth);
}
```

Method name: callback
Method signature: (I)V

Code uses `CallVoidMethod` because return type of callback method is void

Error handling

- ◆ Two difficult areas for interoperability
 - Memory management
 - Error handling
- ◆ JNI native methods can catch, throw exceptions

JNI: Summary

- ◆ Allows Java methods to be implemented in C/C++
- ◆ Interface determined by native method signature
- ◆ Tools generate C interfaces and marshaling code
- ◆ References are treated abstractly, which facilitates memory management
- ◆ Environment pointer provides access to JVM services such as object creation and method invocation

SWIG

- ◆ Tool to make C/C++ libraries easily available in many high level languages:

```
Go Python Perl C/C++ Java Ruby Haskell PHP OCaml F#E
C# Haskell Haskell Haskell Haskell Haskell Haskell ...
```

- ◆ **Goal:** Read interface from C/C++ headers, requiring annotations only to customize.
- ◆ **Marshaling policy:** references treated opaquely. C library must provide extra functions to allow high-level language to manipulate.

www.swig.org

Interface Definition Languages

- ◆ IDLs provide some control over marshaling policies:
 - Are parameters **in**, **out**, or both?
 - Is **NULL** a distinguished value?
 - Should payload of pointers be copied or left abstract?
 - Is **char*** a pointer to a character or a string?
 - Is one parameter the length of an argument array?
 - Who is responsible for allocating/deallocating space?
- ◆ Language-specific IDL compilers generate
 - header files
 - stubs
 - glue code for marshaling/unmarshaling

IDLs

- ◆ Typically look like C/C++ header files with additional declarations and attributes

```
int foo([out] long* l,
        [string, in] char* s,
        [in, out] double * d);
```

- ◆ Annotations tell high-level language how to interpret C/C++ parameters
- ◆ Unlike SWIG, pointers don't have to be abstract on high-level language side
- ◆ Unlike JNI, pointers don't have to be abstract on C side

IDLs: Pointer Annotations

- ◆ Some annotations to clarify role of pointers:
 - **ref**: a unique pointer that can be safely marshaled
 - **unique**: just like **ref** except it may also be **null**
 - **ptr**: could be shared, could point to cyclic data; can't be marshaled
 - **string char***: null terminated sequence of characters, should be treated like a string
 - **size_is(parameter_name)**: pointer is array whose length is given by **parameter_name**

```
void DrawPolygon  
([in, size_is(nPoints)] Point* points  
 [out] int* nPoints)
```

Examples of IDL-based Systems

- ◆ Simple high-level language to C bindings:
 - **camldl**, **H/Direct**, **mlidl**, etc.
- ◆ COM: Microsoft interoperability platform
- ◆ CORBA: OMG's interoperability platform

COM and CORBA both leverage the idea of IDLs to go well beyond simple interoperability, supporting distributed *components*: collections of related behaviors grouped into objects

COM: Component Object Model

- ◆ Purpose (marketing page)
 - "COM is used by developers to create re-usable software components, link components together to build applications, and take advantage of Windows services. ..."
- ◆ Used in applications like Microsoft Office
- ◆ Current incarnations
 - COM+, Distributed COM (DCOM), ActiveX Controls
- ◆ References
 - Don Box, Essential COM
 - MS site: <http://www.microsoft.com/com/>

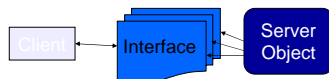
COM

- ◆ Each object (*server*) supports multiple interfaces
 - Each interface gives a different view of the object
- ◆ Interface provides operations on object
 - COM client acquires pointers to an object's interface
 - Invoke methods through pointer as if local object
 - All COM objects provide **QueryInterface** method to support dynamic interface discovery



Versioning

- ◆ MS uses multiple interfaces to support versioning
- ◆ Objects keep existing interfaces, add new ones
 - New client code asks for newer server interfaces
 - Legacy code can continue to ask for older versions

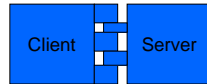


Binary Compatibility

- ◆ Interfaces of COM objects described in MIDL
- ◆ Object impl must conform to C++ vtable layout
 - Each object can be implemented in any language as long as compiler for language can produce vtables
- ◆ Language-specific IDL compiler generates proxy/stub functions for marshaling and unmarshaling to a wire format

Execution Model, Local

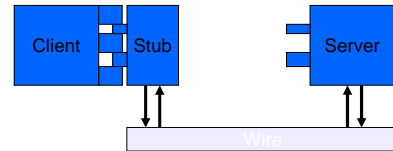
- ◆ If executing in the same address space, client and server objects are dynamically linked.



- ◆ The first time a message is sent to server, code in initial stub vtable finds and loads code, replacing itself with the actual vtable.

Execution Model, Remote

- ◆ If executing in different address spaces, stub vtable marshals arguments, sends message to remote object, waits for response, unmarshals it and delivers it.



COM: Grid Example

- ◆ Grid server object maintains two dimensional array of integers
- ◆ Supports two groups of methods:



IGrid1

```
get() : gets value stored at argument location.
set() : sets value at argument location.
```

IGrid2

```
reset() : resets value of all cells to supplied value.
```

COM: Grid Example IDL

- ◆ Portion of IDL file to describe IGrid1 interface:

```

// Guid and definition of IGrid1
object,
guid(1279b284-c0c5-11d0-8a0b-00a0b90b693d),
helpstring("IGrid1 interface"),
pointer_default(unique)

interface IGrid1 : IUnknown
{
import "unkown.idl";
HRESULT get([in] SHORT n, [in] SHORT n, [out] LONG *value);
HRESULT set([in] SHORT n, [in] SHORT n, [in] LONG value);
};

```

- ◆ Each interface has a globally unique GUID and extends the **IUnknown** interface, which provides **queryInterface** and reference counting methods.

COM: Grid Example Client Code

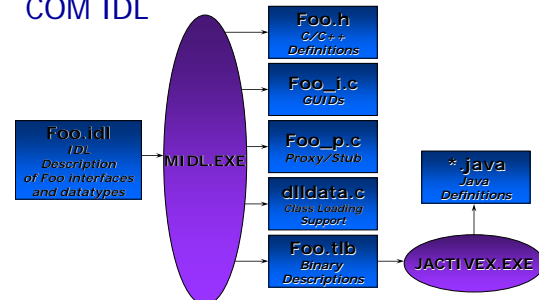
```

#include "grid.h"
void main(int argc, char**argv) {
    IGrid1 *pGrid1;
    IGrid2 *pGrid2;
    LONG value;
    CoInitialize(NULL); // initialize COM
    CoCreateInstance(CLSID_CGrid, NULL, CLSCTX_SERVER,
        IID_IGrid1, (void**) &pGrid1);
    pGrid1->get(0, 0, &value);
    pGrid1->QueryInterface(IID_IGrid2, (void**) &pGrid2);
    pGrid1->Release();
    pGrid2->reset(value);
    pGrid2->Release();
    CoInitialize();
}

```

my.execpc.com/~gopalan/misc/compare.html

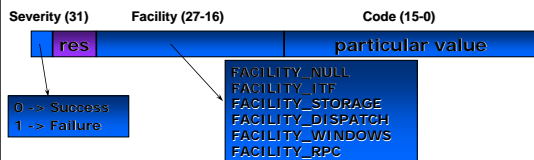
COM IDL



- ◆ COM interfaces may be defined in COM IDL
- ◆ IDL compiler generates C/C++ interface definitions

COM and Error Handling

- ◆ COM (today) doesn't support typed C++ or Java-style exceptions
- ◆ All (remotable) methods must return a standard 32-bit error code called an HRESULT
 - Mapped to exception in higher-level languages
 - Overloaded to indicate invocation errors from proxies



Reference counting

- ◆ Leverage indirection through reference object
 - Clients "Delete" each reference, not each object
- ◆ Object checks references to it
 - Objects track number of references and auto-delete when count reaches zero
 - Requires 100% compliance with ref. counting rules
- ◆ Client obligations
 - All operations that return interface pointers must increment the interface pointer's reference count
 - Clients must inform object that a particular interface pointer has been destroyed

COM Summary

- ◆ Object servers are abstract data types described by interfaces
- ◆ Object servers can be loaded dynamically and accessed remotely
- ◆ Clients interrogate server objects for functionality via RTTI-like constructs (ie, **queryInterface**)
- ◆ Clients notify server objects when references are duplicated or destroyed to manage memory
- ◆ Supports binary-compatible multi-language programming



CORBA

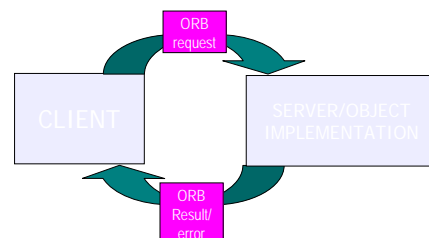
- ◆ Interoperability where systems can't be tightly coupled:
 - Companies working together (telecommunications, medical, etc.)
 - Large system integrations
- ◆ Can't enforce same language, same OS, or same hardware.
 - Engineering tradeoffs, cost effectiveness, legacy systems

OMG

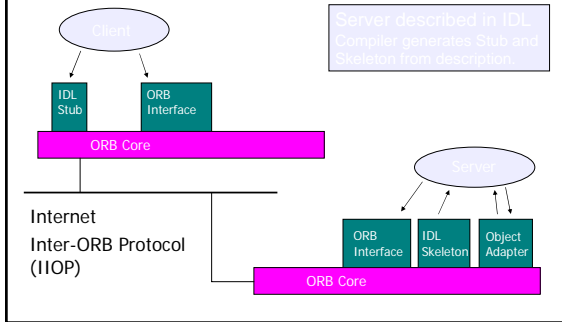
- ◆ CORBA is a standard developed by the Object Management Group.
 - Common Object Request Broker Architecture
 - Over 700 participating companies
 - Request for proposal process
- ◆ Example:
 - Telecommunications industry uses CORBA to manage provisioning process, in which competitors have to work together.

CORBA Concept

- ◆ Insert "broker" between client and server, called the Object Request Broker

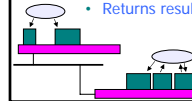


CORBA Architecture



Functions of ORB

- ◆ Communication between client and server
 - Insulates application from system configuration details
- ◆ Local ORB
 - Intercepts calls via stub code
 - Locates server object host machine
 - Sends message with wire representation of request.
- ◆ Remote ORB/Object Adaptor
 - Finds server object implementation, potentially starting new server if necessary, and delivers message.
 - Returns results or error messages to local ORB



CORBA: Grid Example IDL



```
interface grid1 {
    long get1(in short n, in short m);
    void set1(in short n, in short m, in long value);
};

interface grid2 {
    void reset(in long value);
};

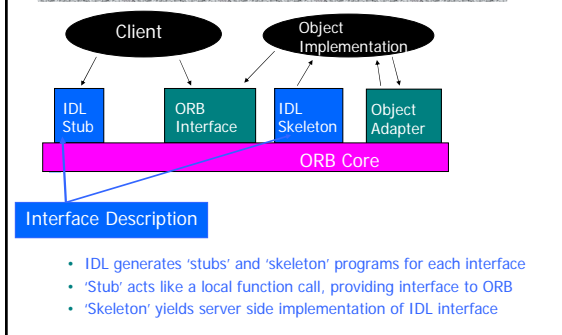
/* multiple inheritance of interfaces
interface grid: grid1, grid2 {}
```

CORBA: Grid Client Code

```
import org.omg.CORBA.*;
import org.omg.CosNaming.*;
import Grid.*;

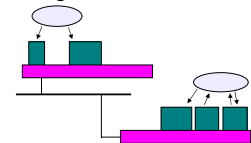
public class GridClient {
    public static void main(String[] args) {
        try {
            ORB orb = ORB.init();
            NamingContext root =
                NamingContextHelper.narrow(
                    orb.resolve_initial_references("NameService"));
            NameComponent name = new NameComponent("GRID");
            grid gridVar = GridHelper.narrow(root.resolve(name));
            value = gridVar.get1(1);
            gridVar.reset(value);
        } catch (SystemException e) {System.err.println(e);}
    }
}
```

Interface description language




CORBA Summary

- ◆ Interoperability for loosely coupled systems
- ◆ Interface definition language specifies server object functionality
- ◆ Language-specific IDL compiler generates stubs and skeletons
- ◆ ORB and related services manage remote message sending



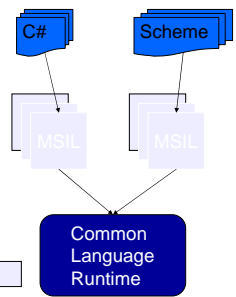
.NET Framework



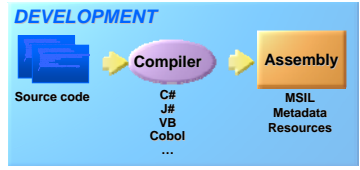
- ◆ Microsoft cross-language platform
 - Many languages can use/extend .NET Framework
 - Compile language to MSIL
 - All languages are conceptually interoperable
- ◆ Focus on security and trust
 - Building, deploy, and run semi-trusted applications
- ◆ Two key components
 - Common Language Runtime
 - .NET Framework Class Library

Current .NET Languages

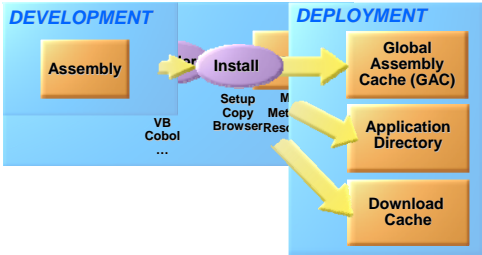
◆ C++	◆ SmallTalk
◆ Visual Basic	◆ Oberon
◆ C#	◆ Scheme
◆ Jscript	◆ Mercury
◆ J#	◆ Oz
◆ Perl	◆ RPG
◆ Python	◆ Ada
◆ Fortran	◆ APL
◆ COBOL	◆ Pascal
◆ Eiffel	◆ ML
◆ Haskell	



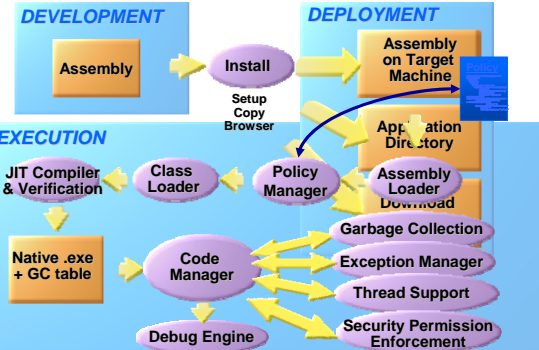
Common Language Runtime



Common Language Runtime



Common Language Runtime



.NET: SQL Program Examples

```

C#
string s = "authors";
SqlCommand cmd = new SqlCommand("select * from " + s, sqlconn);
cmd.ExecuteReader();

C++
String *s = "authors";
SqlCommand cmd = new SqlCommand(
    String::Concat("select * from ", s),
    sqlconn);
cmd.ExecuteReader();
  
```

.NET: SQL Program Examples

Perl

```
string $s = "authors";
SqlCommand cmd = new SqlCommand(
    string::Concat("select * from ", $s),
    sqlconn);
cmd.ExecuteReader();
```

Python

```
s = "authors"
cmd = SqlCommand("select * from " + s, sqlconn)
cmd.ExecuteReader();
```

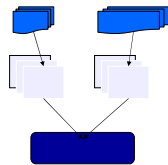
.NET: SQL Program Examples

COBOL

```
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
REPOSITORY.
    CLASS SqlCommand AS "System.Data.SqlClient.SqlCommand"
    CLASS SqlConnection AS "System.Data.SqlClient.SqlConnection".
DATA DIVISION.
WORKING-STORAGE SECTION.
    PI PIC X(255).
    PI cmd-string PIC X(255).
    PI cmd OBJECT REFERENCE SqlCommand.
    PI sqlconn OBJECT REFERENCE SqlConnection.
PROCEDURE DIVISION.
    ** Establish the SQL connection here somewhere.
    MOVE "authors" TO str.
    STRING "select * from " DELIMITED BY SIZE,
    str DELIMITED BY + INTO cmd-string.
    MOVE SqlCommand "cmd" USING BY VALUE cmd-string sqlconn RETURNING cmd
    MOVE cmd "ExecuteReader".
```

.NET Interoperability

- ◆ As examples illustrate, language implementers make CLR Framework Class Hierarchy available within language.
- ◆ Compilers can record meta data along with MSIL code.
- ◆ Other languages can read meta data to use compiled code from other languages.
- ◆ Requires cooperation between compiler writers.



Security Issues

- ◆ OS security is based on user rights
- ◆ CLR security gives rights to code

Trusted user Untrusted code	Trusted user Trusted code
Untrusted user Untrusted code	Untrusted user Trusted code

.NET Summary

- ◆ Compile multiple languages to common intermediate language (MSIL) which serves as lingua franca instead of C/C++.
- ◆ MSIL executed by virtual machine
 - Similar to Java VM in many respects
 - More elaborate security model
 - JIT is standard, instead of interpreter
- ◆ MSIL contains special provisions for certain languages.

Summary



- ◆ Interoperability is a difficult problem, with lots of low-level details
- ◆ C/C++ can serve as a lingua franca
- ◆ Interface definition languages specify interfaces between components
- ◆ IDL compilers can generate marshaling code
- ◆ COM and CORBA leverage IDLs to support distributed computation
- ◆ .NET's MSIL and CLR can serve as a higher level lingua franca