1. Background and terminology
2. Interfaces between systems
3. Native-side system $\Rightarrow$ Java method
4. Java application $\Rightarrow$ Native-side system
5. General comments

1. Background and terminology
2. Interfaces between systems
3. Native-side system $\Rightarrow$ Java
4. Java application $\Rightarrow$ Native-side system
5. General comments

Java

- a programming language, in the C family, but in many ways more advanced than C or C++;
- "object-oriented"; specifically, programs are built from methods contained in classes, not from functions — `x.diag()` for the diagonal of `x`, not `diag(x)`
- designed to support distributed computing on the web;
- a large and growing set of tools, notably for interacting with users, but for *many* other areas as well.

Moral: a major part of our future, especially for delivering our technology to users or in packages.

Java — Some Terms

package the large-scale software unit, e.g., `org.omegahat.NativeInterface.S`;

class the small-scale software unit, within a package, e.g, `SInterfaceManager`;

method defined within a class — the only "call" within Java is to a method, e.g., `toS`;

application a class that can be run as a command, e.g., a GUI or interactive language, e.g., `omega` or `jpython`.

Java and Interfaces
(more terms)

Java Native Interface (JNI) a facility for implementing Java methods in C or C++, *and* for embeding Java within a C or C++program.

native-side system (my term) any suitable language & system to link with Java (must have access to C), e.g., S, R, or Lisp-stat.

embed to run a system that normally interfaces to the user, but from *within* another system.

The Omega Project

- A joint project to work towards the next generation of statistical systems, emphasizing Java and distributed computing (but not restricted to that).
- Goals include: Open software; wide participation; high-quality software.
- Current areas: interactive environment (language & tools); packages under discussion or development for models, graphics, and interfaces.

1. Background and terminology
2. Interfaces between systems
3. Native-side system $\Rightarrow$ Java
4. Java application $\Rightarrow$ Native-side system
5. General comments

Inter-System Interfaces

- Two computing systems, each with significant “structure”; e.g., general user interaction, a programming language & environment, general data structures, and/or object management.
- Users are working in one such system and want to use another for some computations.
- Traditional approach: Invoke function/method in a foreign system; from a current language and system.

Issues

- Why?
  To use existing, tested software rather than re-implementing it, particularly for tough jobs.

  To use a system in a computational area not covered by the current system.
- Why not?
  Communication/synchronization problems, data conversion, possibly extra overhead.
- New approaches: distributed computing, more general control.
- Part of smoothing the evolution to new systems.

General (and Traditional) Idea

- From the current system, choose a single command (or function, or method, or subroutine) in the target system;
- Construct data in the target system for the arguments to the command;
- Invoke the command and recover the computed results.

Examples

1. $\{$ S, or R, or Matlab $\} \Rightarrow \{$ a C or Fortran routine $\}$;
2. $\{$ Perl, or S, or $\cdots$ $\} \Rightarrow$ a shell command;
3. Java $\Rightarrow$ database management systems (the JDBC).
4. Java $\Leftrightarrow$ C (the Java native interface).

The Omega Interface Manager

- Package `org.omegahat.NativeInterface` has classes, methods, and C tools for native-side $\Leftrightarrow$ Java interfaces. The $\Rightarrow$ part will usually need specializing to a particular native-side system; the $\Leftarrow$ part always will.

- This talk describes
  - general tools to extend the Native Interface
  - a specialization of the native side of the interface to S or S-Plus.

- Other specializations, e.g., to R or Lisp-stat, should work similarly.

1. Background and terminology
2. Interfaces between systems
3. Native-side system $\Rightarrow$ Java method
4. Java application $\Rightarrow$ Native-side system
5. General comments

C $\Rightarrow$ Java

The `InterfaceManager` Java class and C tools provide for 4-step interface:

1. Start the Java virtual machine;
2. Convert native-side objects to Java and store them in a Java database;
3. Find and run a Java method (using classes and methods from the Omega environment) using objects in the database.
4. Convert Java database objects back to C data;

Native-Side Systems $\Rightarrow$ Java

Interactive statistical systems (e.g., S, R, and Lisp-stat) can specialize the interface manager by extending the `InterfaceManager` class in Java and providing access to the C tools from the statistical system.

Data conversion between the structures of the native-side system and Java or a Java-based system is a fundamental limitation.

S $\Rightarrow$ Java

Specialize steps 1-4 using S.

1. Run S with access to Java; start the JVM configured to give access from S.

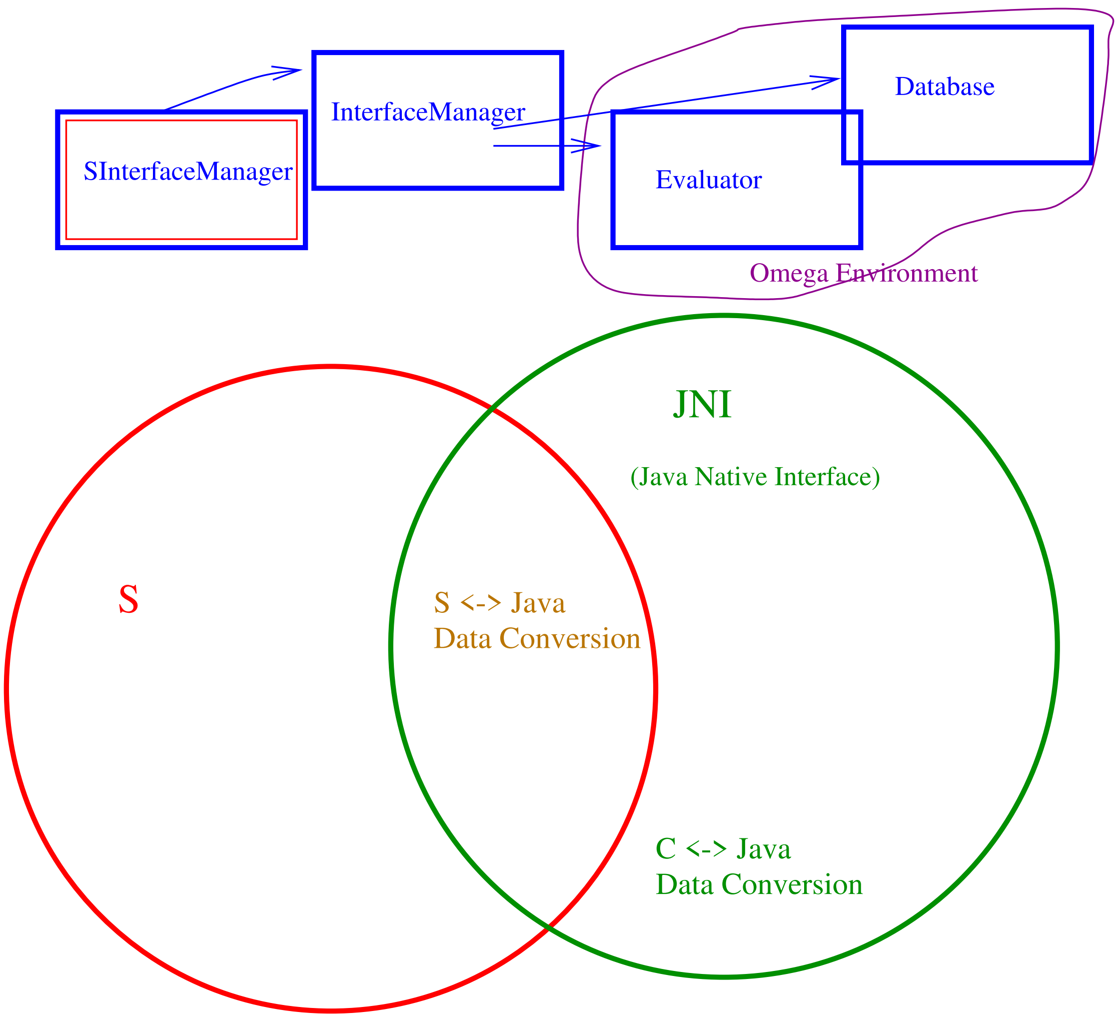
2. Convert S objects to Java objects, providing the target class when needed.

3. Use the `.Java` function in S to call a Java method.

4. Convert objects in the Java database back to S, *when needed*.

For the eventual user in S, the details will be hidden by S functions and classes.

S ⇔ Java



S -> Java, via the Java Native Interface, the InterfaceManager class, and Omega tools.

S $\Rightarrow$ Java
Example: Database Query

Ingredients:

1. Data in a database (say, Oracle); queries in the SQL allow us to extract very general (& maybe very large) subsets of the data.

2. Java (via JDBC) has a general interface to this that lets us work with the data a few rows at a time, and uniformly over different database systems.

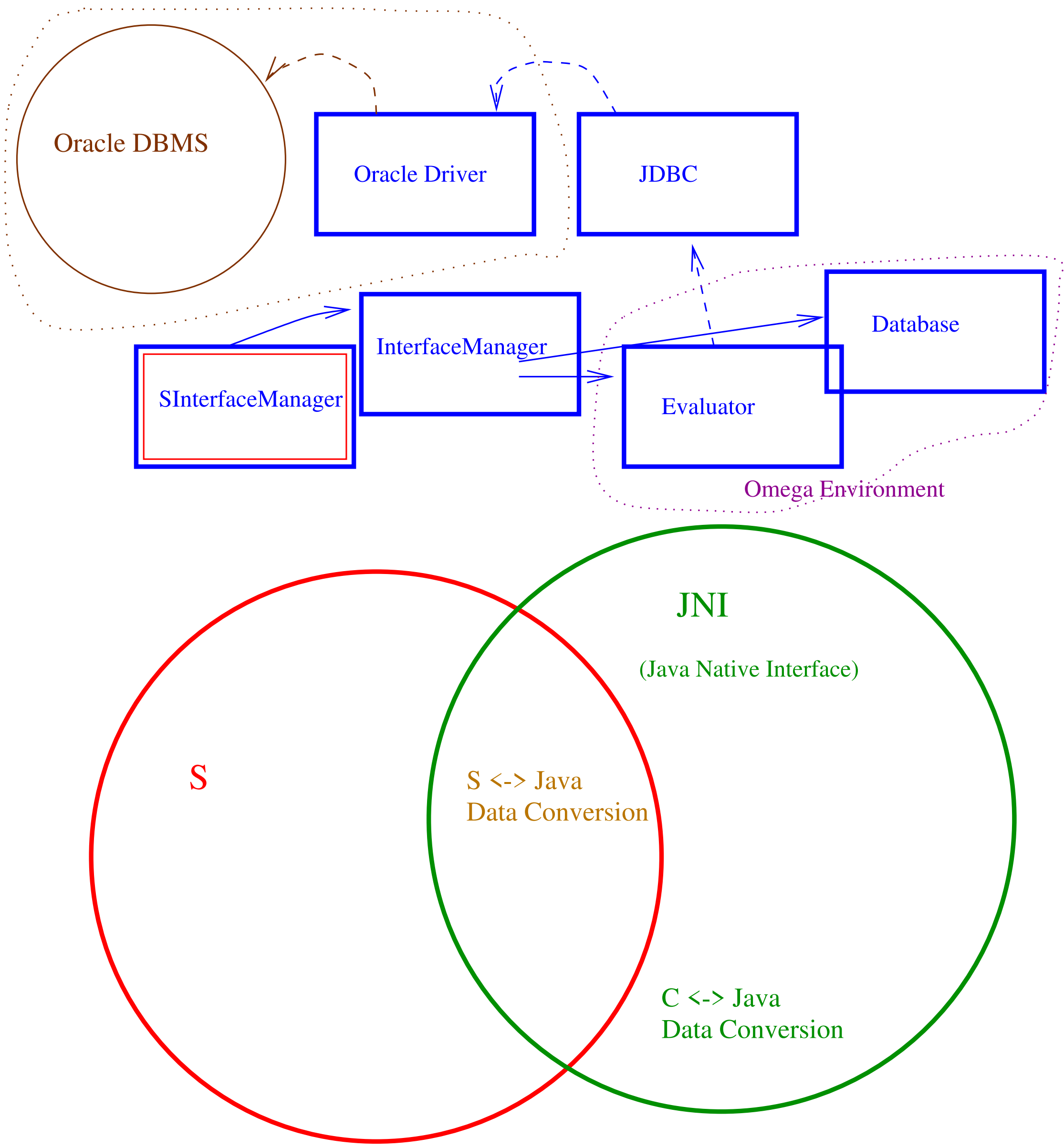
3. The S $\Rightarrow$ Java interface lets us do the query and the successive rows in Java, then bring data into S in steps.

4. (not essential, but nice) `DataWindow` Java class simplifies dealing with blocks of rows, returning “data-frame” style object.

What the User Sees

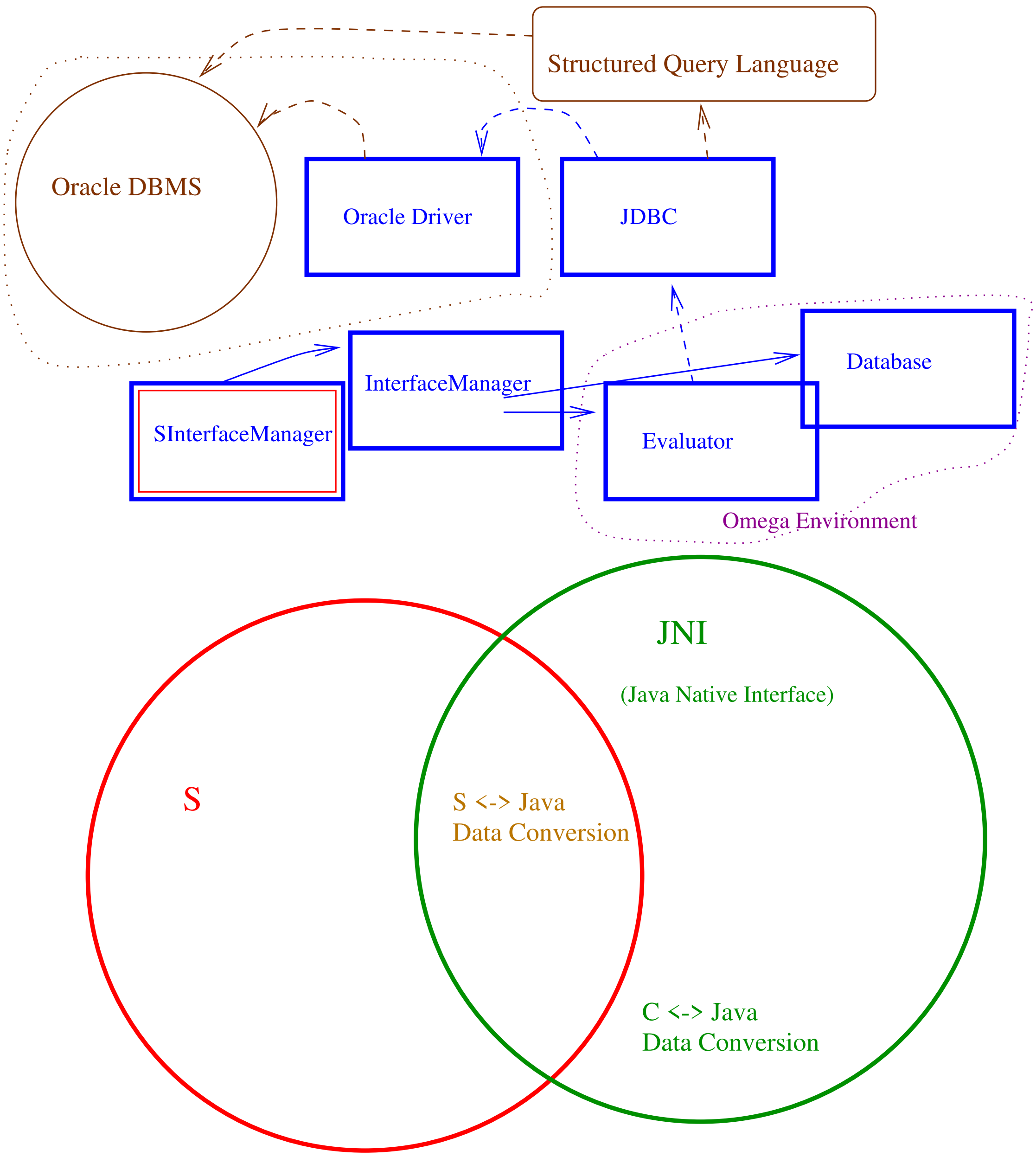
```
> db = jdbcDatabase()
> con = jdbcConnection(db,
+  "jdbc:oracle:thin:@apollo:1521:oras", "opto")
> result = jdbcSelect(db, "call_history",
+  c("DURATION", "ORIGLOGIN", "TALKTIME"))
> xx = dataWindow(result, get = T)
> class(xx)
[1] "named"
> names(xx)
[1] "DURATION"  "ORIGLOGIN" "TALKTIME"
> xx[[1]][1:10]
 [1]  201  521 1462  298  523   31    1 1288 1266  697
> xx[[2]][1:10]
 [1] "" "" "" "" "" "" "" "" "" ""
> xx[[3]][1:10]
 [1]  198  511 1452  289  513   29    0 1280 1256  687
```

S ⇔ Java & JDBC



S -> Java, via the Java Native Interface, the InterfaceManager class, and Omega tools.

S ⇔ Java & JDBC & SQL



S -> Java, via the Java Native Interface, the InterfaceManager class, and Omega tools.

Programming for S $\Rightarrow$ Java

```
getDataWindowVariable =
## return variable ‘which’ (either the
## name or the index) from the ‘dataWindow’
## Java object.
function(dataWindow, which) {
  if(is(which, "character"))
    sigs = c("", "String")
  else {
    which = as(which, "integer")-1
    sigs = c("", "I")
  }
  var = .Java("getVariable", dataWindow,
              which, sigs=sigs)
  var = .Java("toArray", var)
  fromJava(var)
}
```

Components of the S $\Rightarrow$ Java Interface

.Java invoke Java methods, static methods, constructors, and methods in the interface manager itself.

toJava convert an S object to Java and store it in the database. (Often needs to specify the Java datatype.)

fromJava get a Java object from the database and convert it to S.

1. Background and terminology
2. Interfaces between systems
3. Native-side system $\Rightarrow$ Java method
4. Java application $\Rightarrow$ Native-side system
5. General comments

Java application $\Rightarrow$

1. From a Java application (typically a GUI or Java-based interactive language),
2. start the native-side system as an embedded system.
3. Send tasks or commands to the system.
4. Convert objets to and from the system as in the $\Rightarrow$ Java case.

Omega ⇒ S: Example

```
Om:  db = new OracleDatabase()
 org.omegahat.Databases.JDBC.OracleDatabase@7f8f1a
Om:  con = db.connect(
    "jdbc:oracle:thin:@apollo:1521:oras",
    "opto", "lasers")
 oracle.jdbc.driver.OracleConnection@1c6e402
Om:  result = db.doQuery(
"select DURATION, ORIGLOGIN, TALKTIME from call_history")
 oracle.jdbc.driver.OracleResultSet@450a60
Om:  xx = new DataWindow(result)
 org.omegahat.Databases.JDBC.DataWindow@ac2dfe
Om:  xx.nextRows(-1)
 null
```

```
Om:  y1 = xx.getVariable(0)
 org.omegahat.Databases.JDBC.DataVariable@1138a25
Om:  y1 = xx.getVariable(0).toArray()
Om:  y1.getClass()
 class [D
Om:  y3 = xx.getVariable(2).toArray()
[D@b68619,
Om:  S = new SInterfaceManager()
 org.omegahat.NativeInterface.S.SInterfaceManager@1ac046f
Om:  S.toS(y1, "Duration")
 null
Om:  S.toS(y3, "Talk")
 null
Om:  S.doSTask("plot(Duration, Talk)")
 null
Om:  y1.length
 16384
```

```
Om:  result = db.doQuery("select * from call_history")
 oracle.jdbc.driver.OracleResultSet@29520d
Om:  rsmd = result.getMetaData()
 oracle.jdbc.driver.OracleResultSetMetaData@ca02c0
Om:  for(int i = 1; i < 55; i++)
     System.out.println(i+": "+
     rsmd.getColumnName(i)+": "+rsmd.getColumnTypeName(i));
 1: CALLID: NUMBER
 2: ACWTIME: NUMBER
 3: ANSHOLDTIME: NUMBER
 4: CONSULTTIME: NUMBER
 5: DISPTIME: NUMBER
 6: DURATION: NUMBER
 ...
 54: CMSID: VARCHAR2
```

1. Background and terminology
2. Interfaces between systems
3. Native-side system $\Rightarrow$ Java method
4. Java application $\Rightarrow$ Native-side system
5. General comments

Other Interface Techniques

CORBA very general, requires interfacing both ends.

PVM, *etc.* manage communication & control, but not data conversion.

home-made (e.g., earlier S/Java) communication & control are questionable.