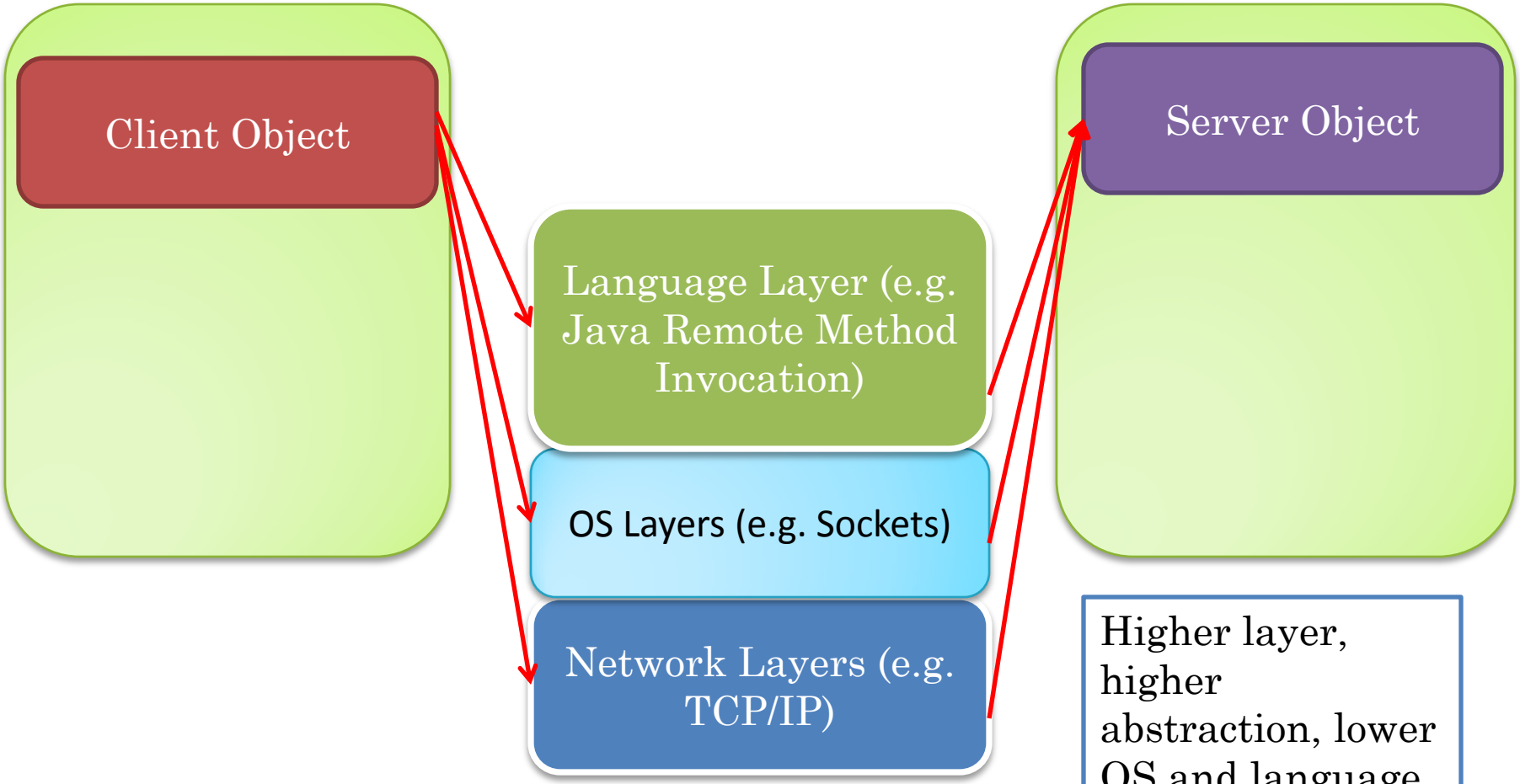
# Java Remote Method Invocation (RMI)

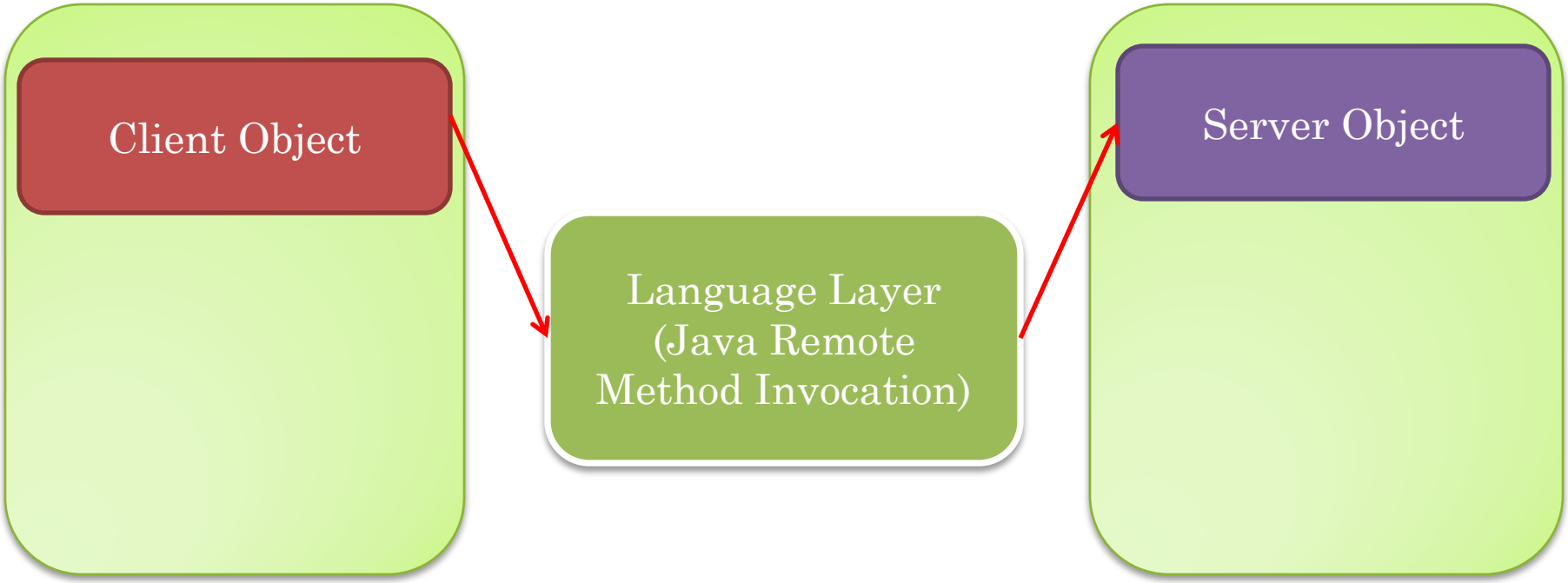
**Prasun Dewan**
**Department of Computer Science**
**University of North Carolina at Chapel Hill**
**dewan@cs.unc.edu**

# Communication Layers




Higher layer, higher abstraction, lower OS and language interoperability.

# Remote Method Invocation



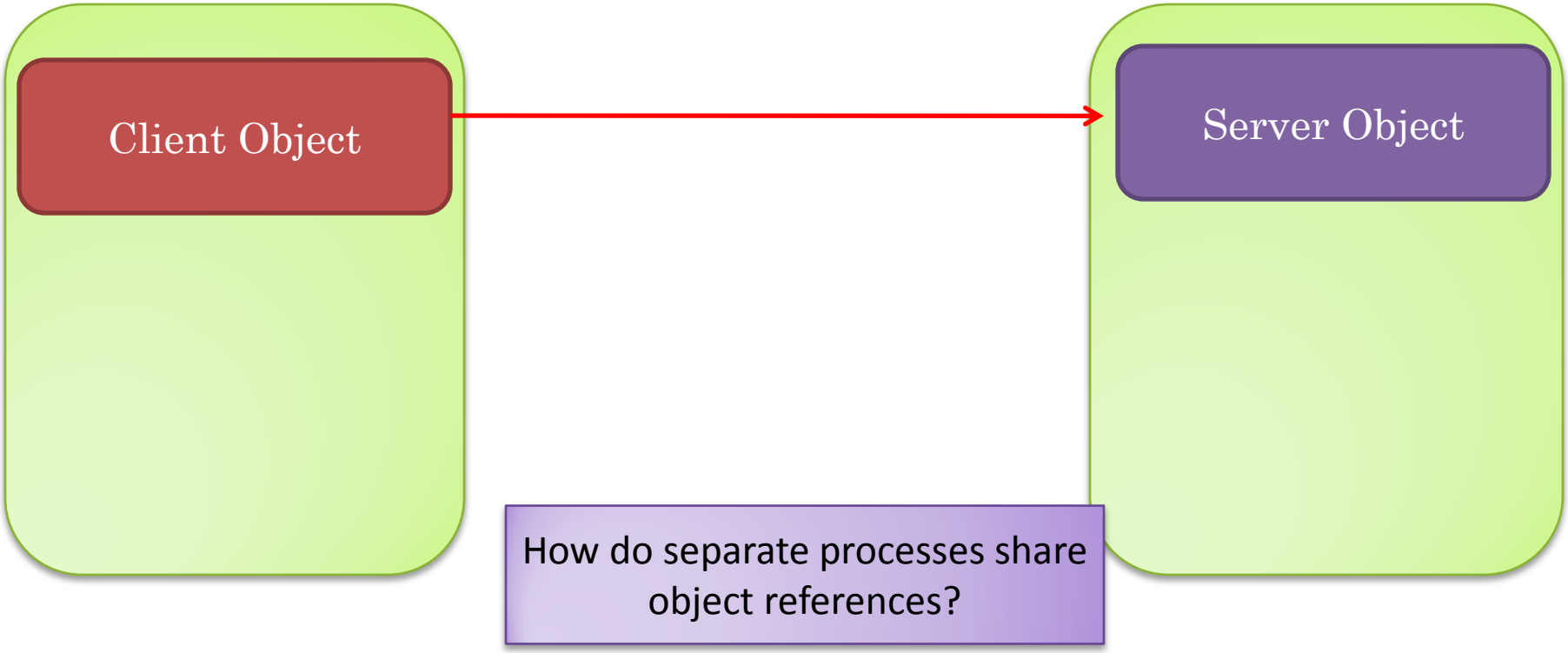
# Counter

```
public interface Counter {
  void increment(int val);
  int getValue() throws RemoteException;
}
```

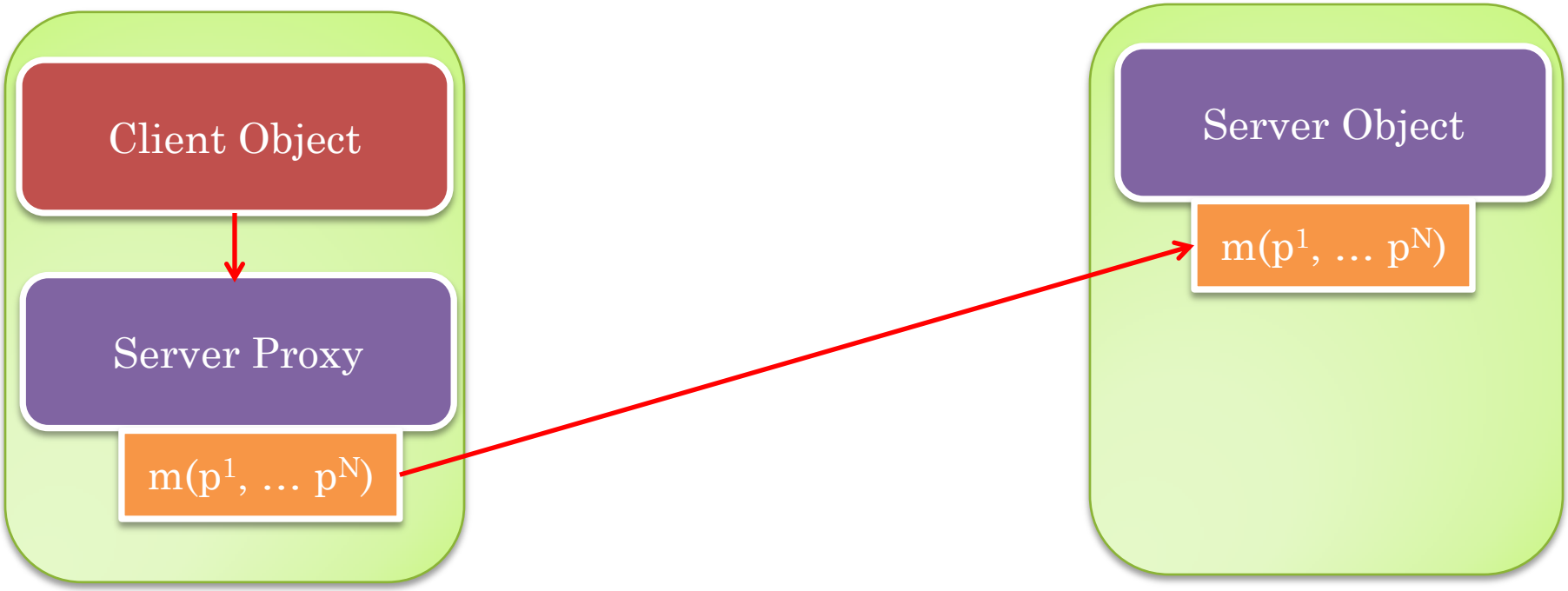
# Counter

```
public class ACounter implements Counter{
  public ACounter() {
    super();
  }
  Integer value = 0;
  public Object getValue() {
     return value;
  }
  public void increment(int val) {
    value += val;
  }
  public String toString() {
    return "Counter:" + value;
  }
  public boolean equals(Object otherObject) {
    if (!(otherObject instanceof Counter))
      return false;
    return getValue() == ((Counter) otherObject).getValue();
  }
}
```

# Remote Method Invocation



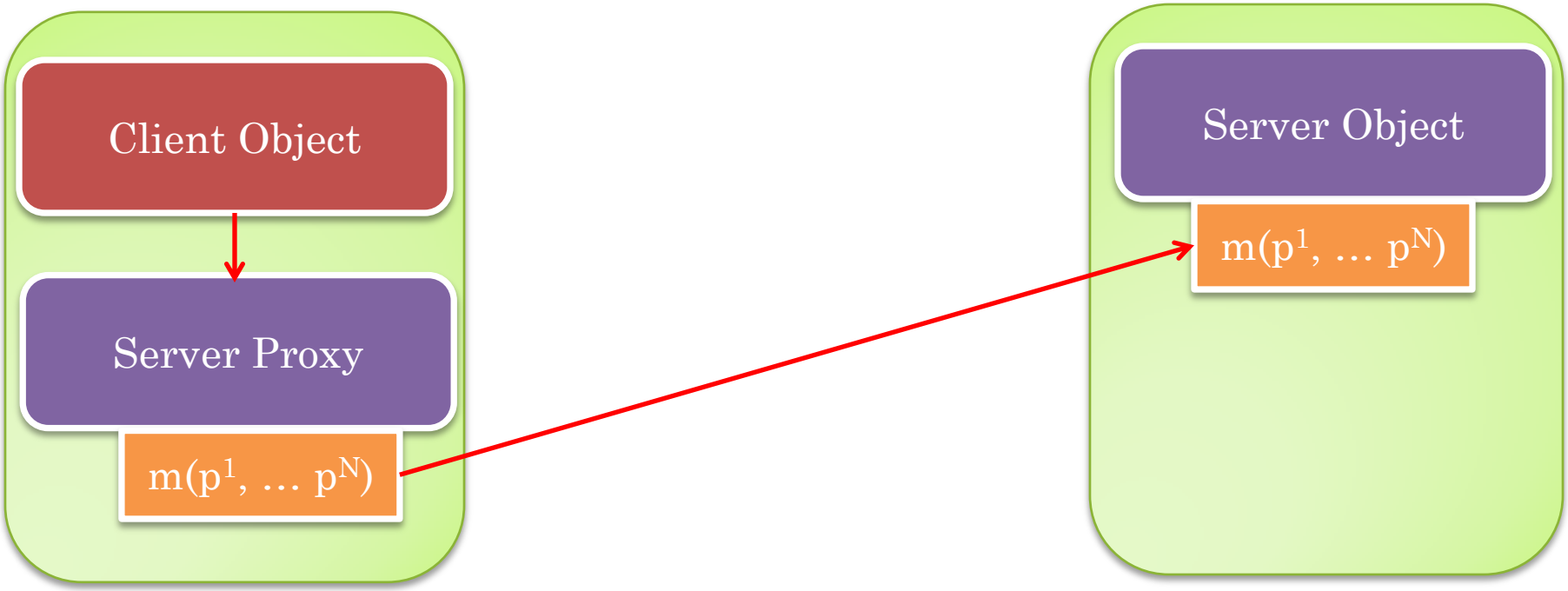
# Local References in Both Address Spaces




Proxy is generate by RMI System.

Proxy and server objects connected through external name

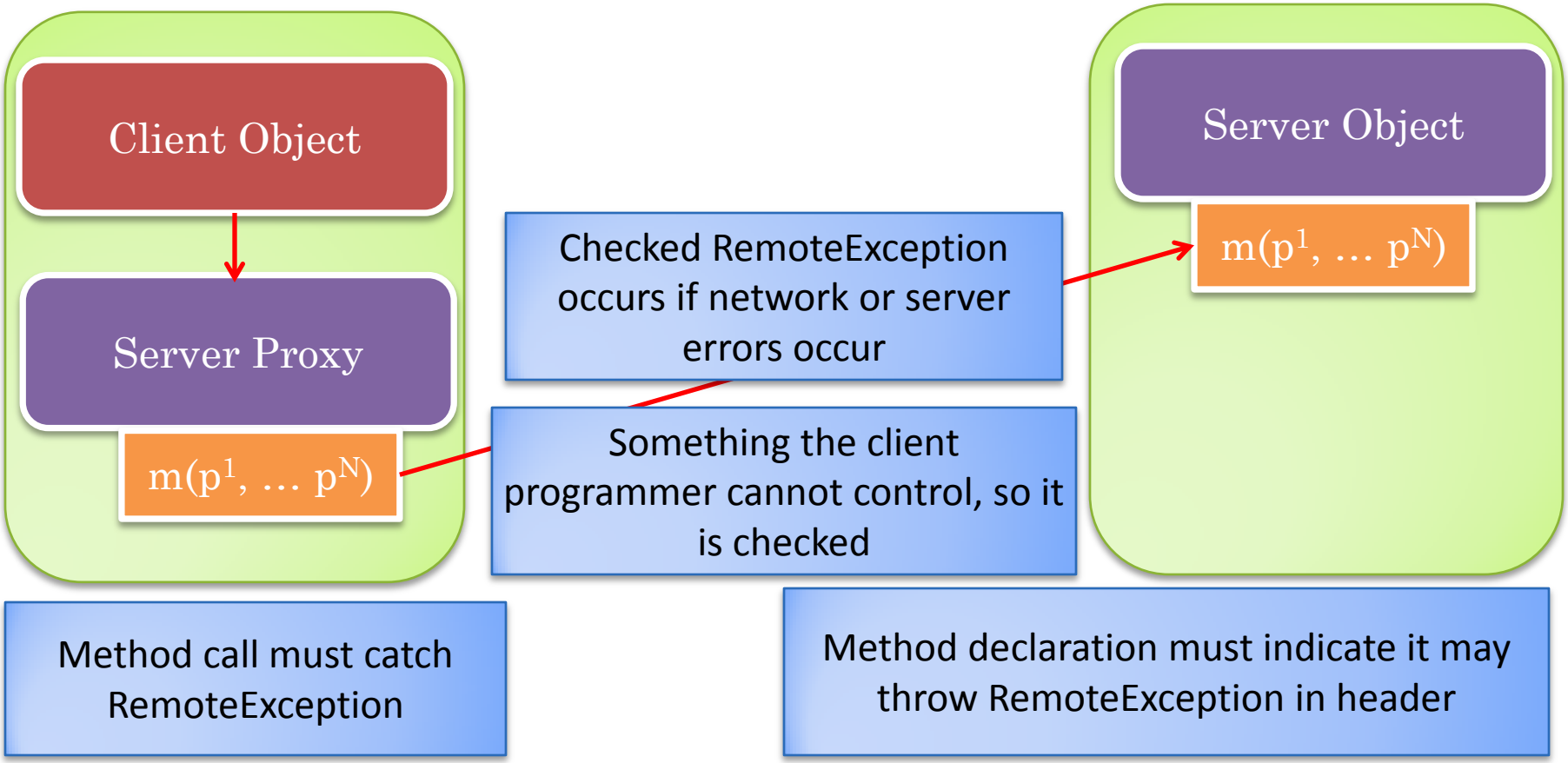
# Local References in Both Address Spaces



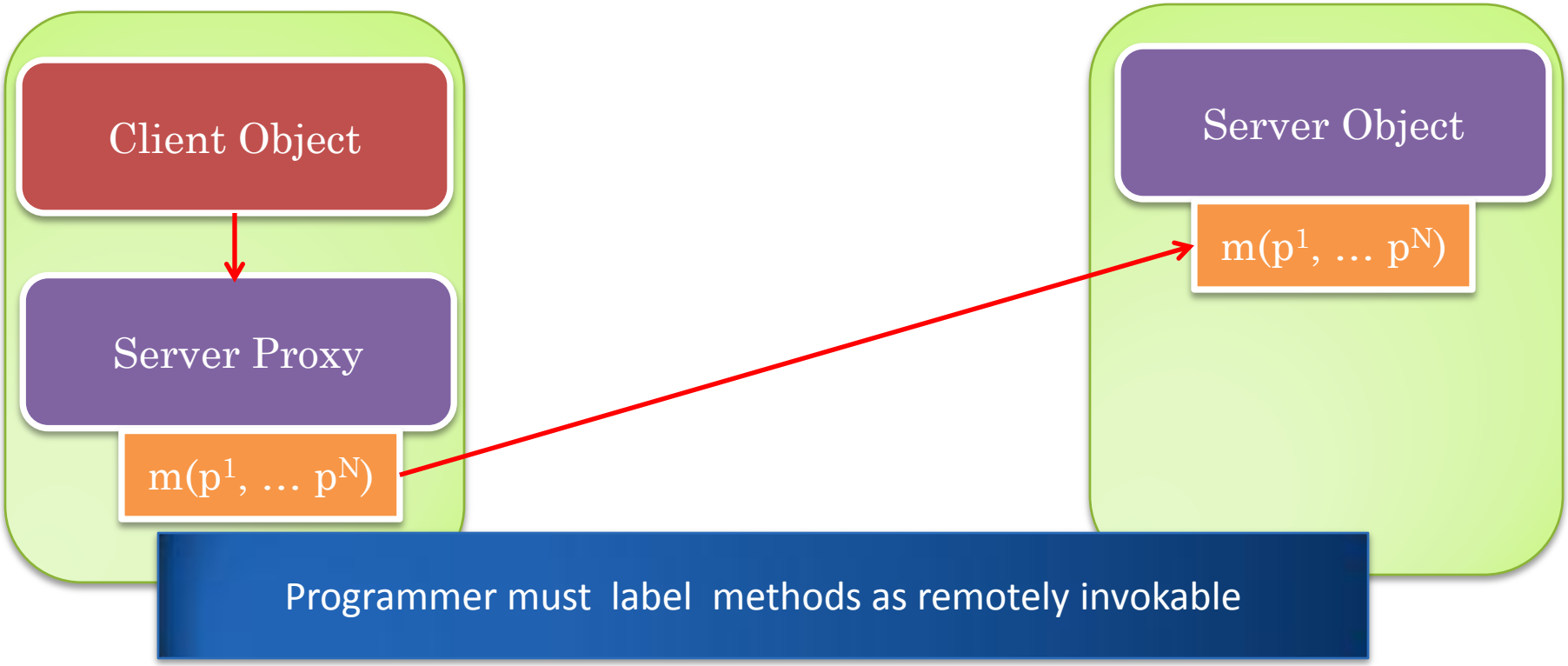

Remote method invocation has the same syntax as local method invocation.

However, caller and callee are distribution-aware

# Checked RemoteException



# Labeling Remote Methods




Programmer must label methods as remotely invokable

If an interface "extends" or a class "implements" the Remote interface, then every method in that class/interface labeled as Remote

Java ensures that every labeled method has the throws clause and generates proxy method for only such a method

# Local vs. Distributed Counter

```
public interface Counter {
  void increment(int val);
  Object getValue();
}
```

```
public interface DistributedRMICounter extends Remote {
  void increment(int val) throws RemoteException;
  Object getValue() throws RemoteException;
}
```

Caller should handle errors

Checked exceptions

Distribution awareness

# ACounter

```
public class ACounter implements Counter{
  public ACounter() {
    super();
  }
  Integer value = 0;
  public Object getValue() {
     return value;
  }
  public void increment(int val) {
    value += val;
  }
  public String toString() {
    return "Counter:" + value;
  }
  public boolean equals(Object otherObject) {
    if (!(otherObject instanceof Counter))
      return false;
    return getValue() == ((Counter) otherObject).getValue();
  }
}
```

Class header need not have throws class as interface is used by client.

DistributedRMICounter does not extend Counter

How is it changed?

# DistributedRMICounter

```
public class ADistributedInheritingRMICounter extends ACounter
implements DistributedRMICounter{
  @Override
  public boolean equals(Object otherObject) {
    if (!(otherObject instanceof DistributedRMICounter))
      return super.equals(otherObject);
    try {
      return getValue().equals(
             ((DistributedRMICounter) otherObject).getValue());
    } catch (RemoteException e) {
      e.printStackTrace();
      return false;
    }
  }
}
```

How to do client specific processing?

Inherited methods implement two different methods, with and without throws clause

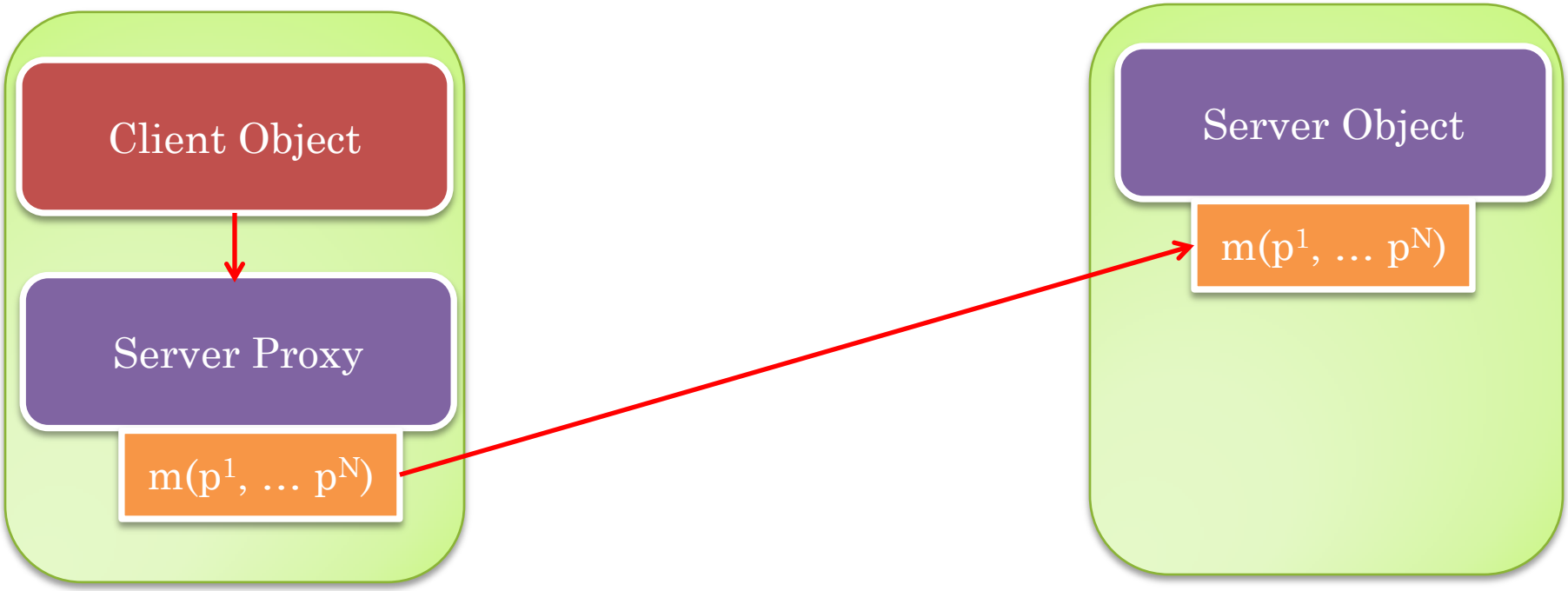
# Caller-Specific Processing

```
public class ADistributedInheritingRMICounter extends ACounter
implements DistributedRMICounter{
  @Override
  public boolean equals(Object otherObject) {
      …
  }
 public int getValue() {
    try {
      System.out.println(RemoteServer.getClientHost());
    } catch (ServerNotActiveException e) {
      e.printStackTrace();
    }
    return super.equals(otherObject);   }
}
```

RemoteServer

static String getClientHost()

# How to create a local proxy




How to connect the two?

Proxy and server objects connected through external name and transfer of serialized proxies

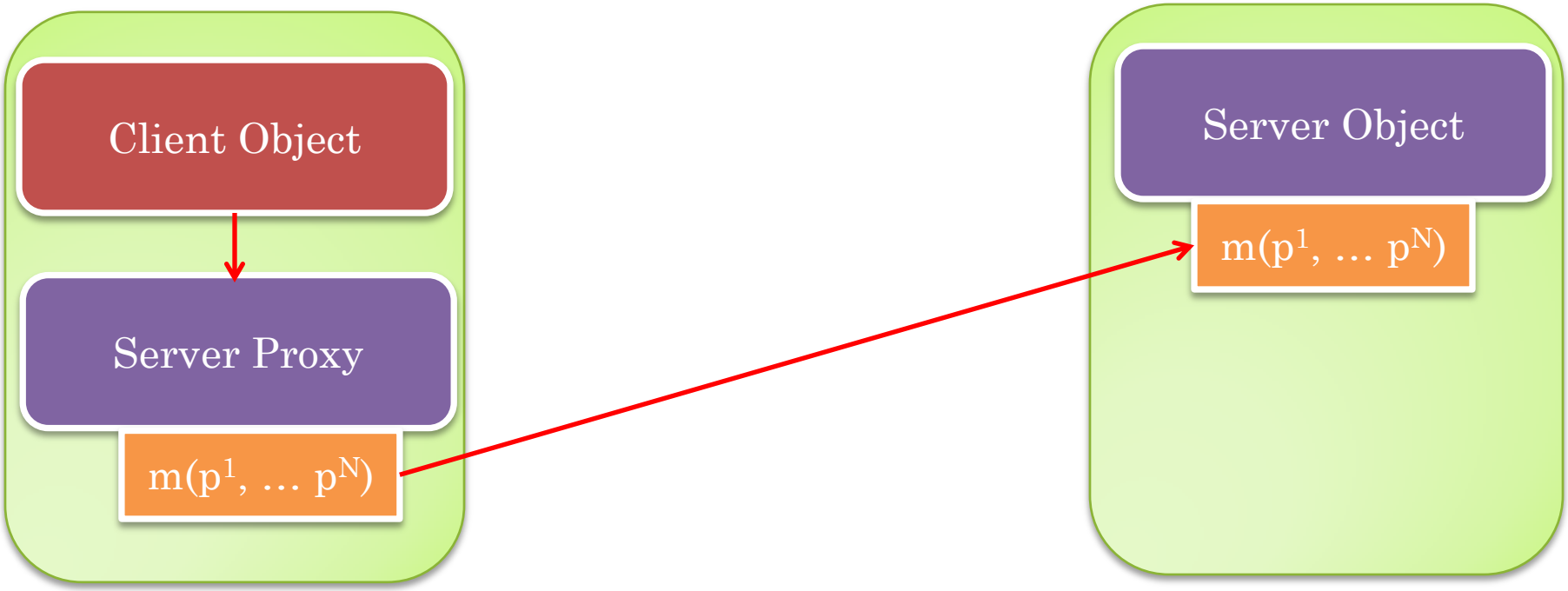
# External Name and Internal Name Binding

| Objects shared among processes have external names. |
| --- |
| file name for files |
| <machine, port> for socket |

| A mechanism is provided to bind a local reference to an external name |
| --- |
| File is opened in write mode giving file name |
| Server socket is bound to <machine, port> |

| A mechanism is provided to bind the external name to local reference |
| --- |
| File is opened in read mode giving file name |
| Clients socket is connected to server socket using <machine, port> |

# Local References in Both Address Spaces

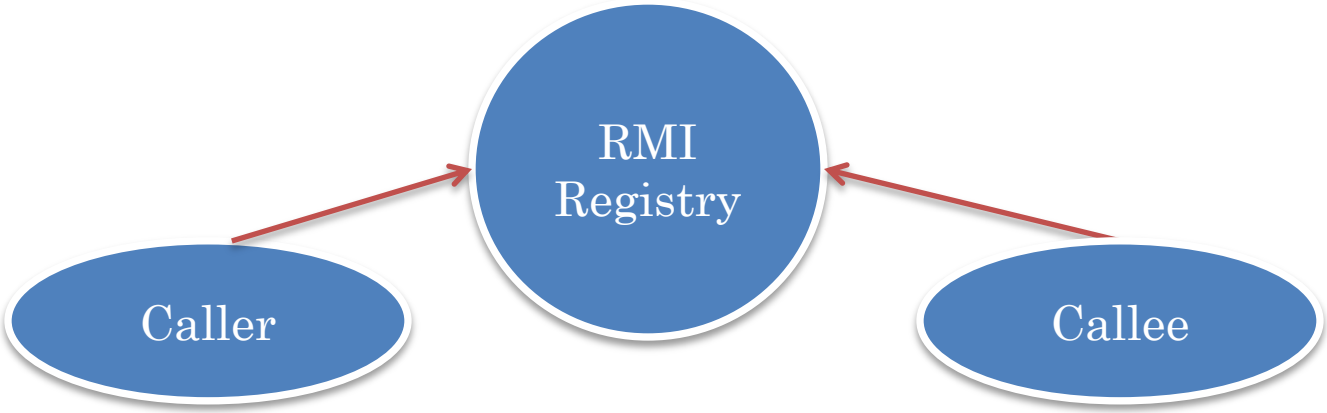



How to connect the two?

Proxy and server objects connected through external name and transfer of serialized proxies

How exactly?

# Name Server

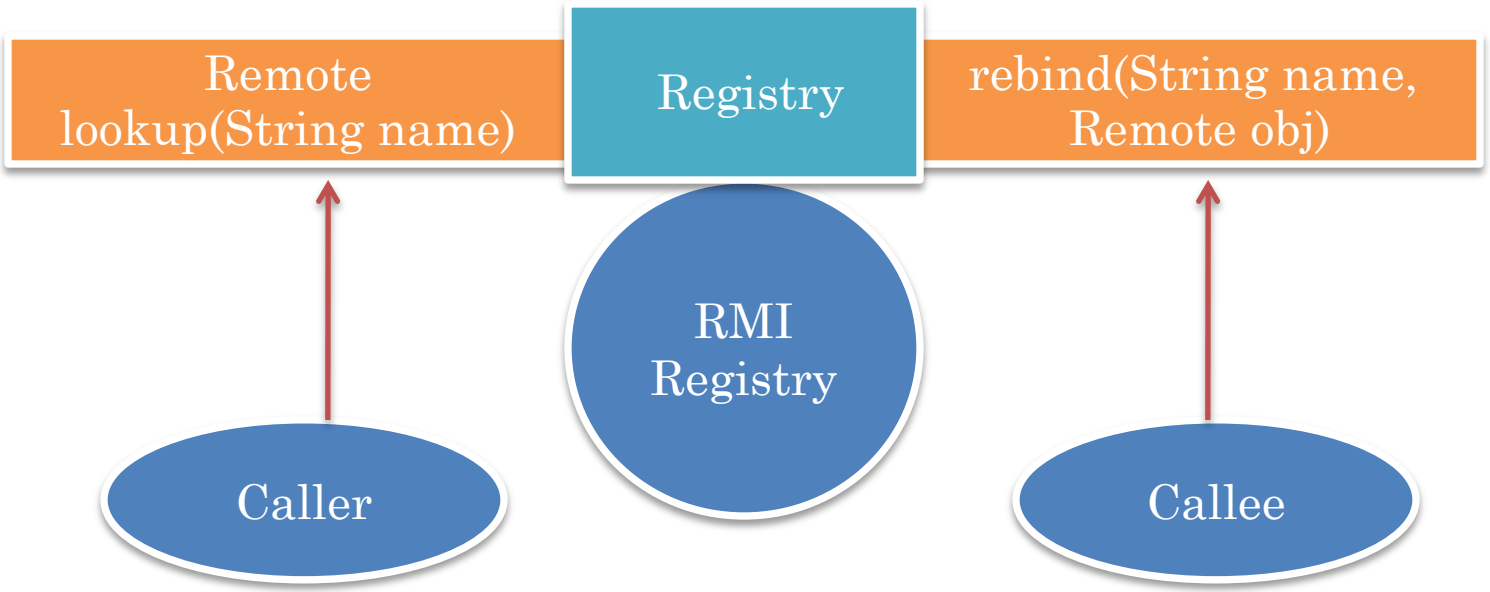



Name server keeps (name, object) pairs

Caller registers (name, proxy) pair

Caller  gets object registered for name

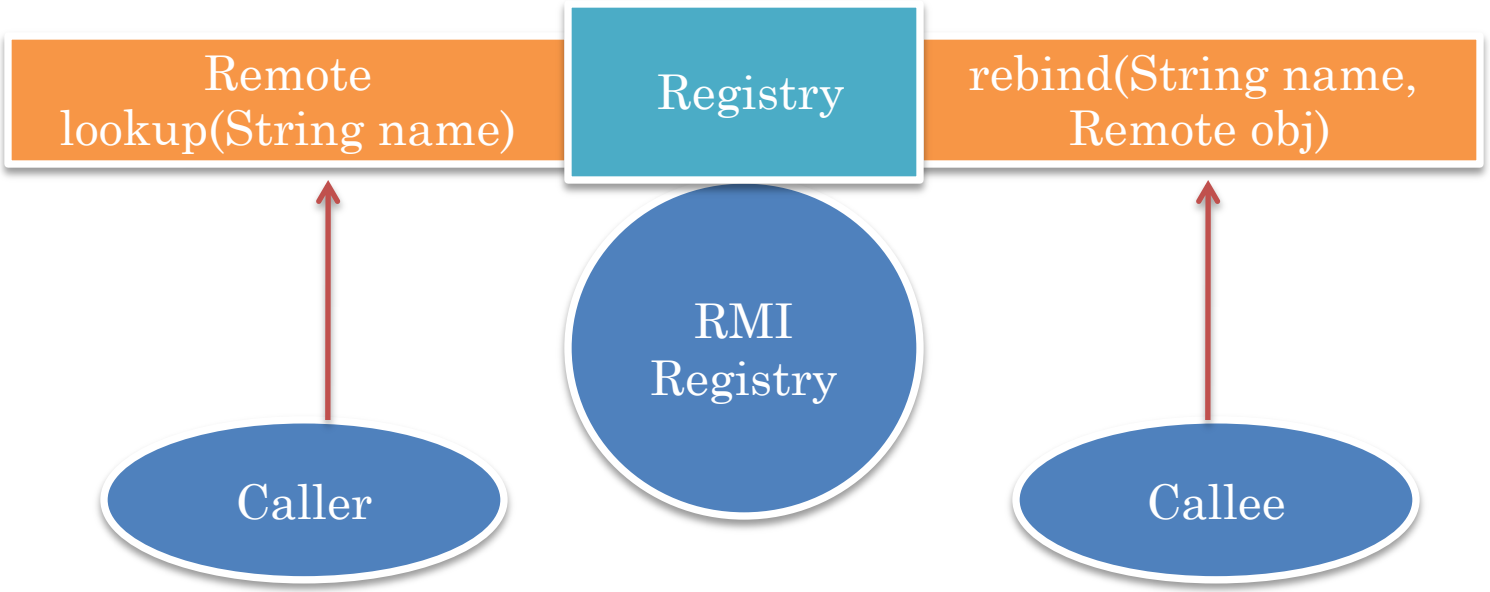
# Name Server Methods




Only instances of Remote can be registered because of special error handling requirement

RMIRegistry must have specific subtype of Remote in path so it can store it in memory

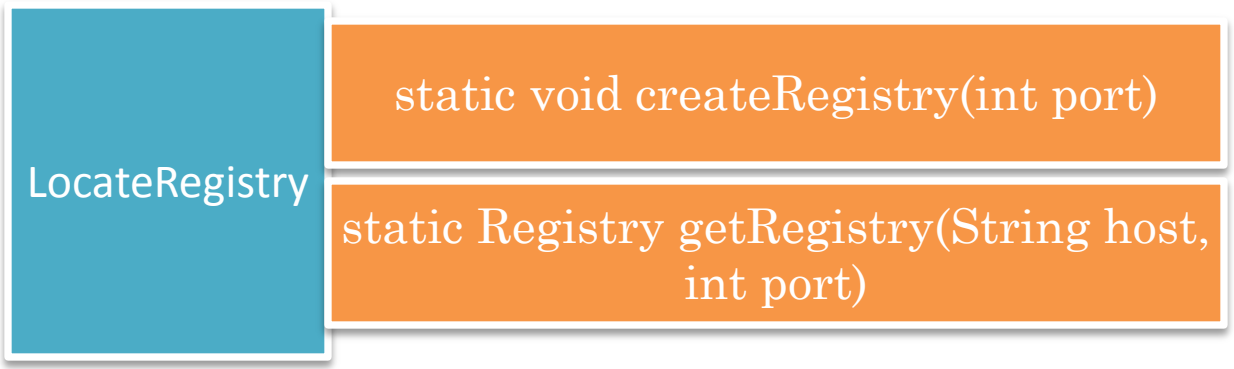
# Name Server Methods




- Name server keeps (name, object) pairs
- How to start RMI Registry Server?
- How to get proxy reference to name server?
- Bootstrapping problem!

# Starting RMI Registry From Console

```
set javabin=D:\"Program Files"\Java\jre1.6.0_03\bin
set CLASSPATH=D:/dewan_backup/java/gipc/bin
%javabin%\rmiregistry 1099
```




RMI Registry is started as part of calling process

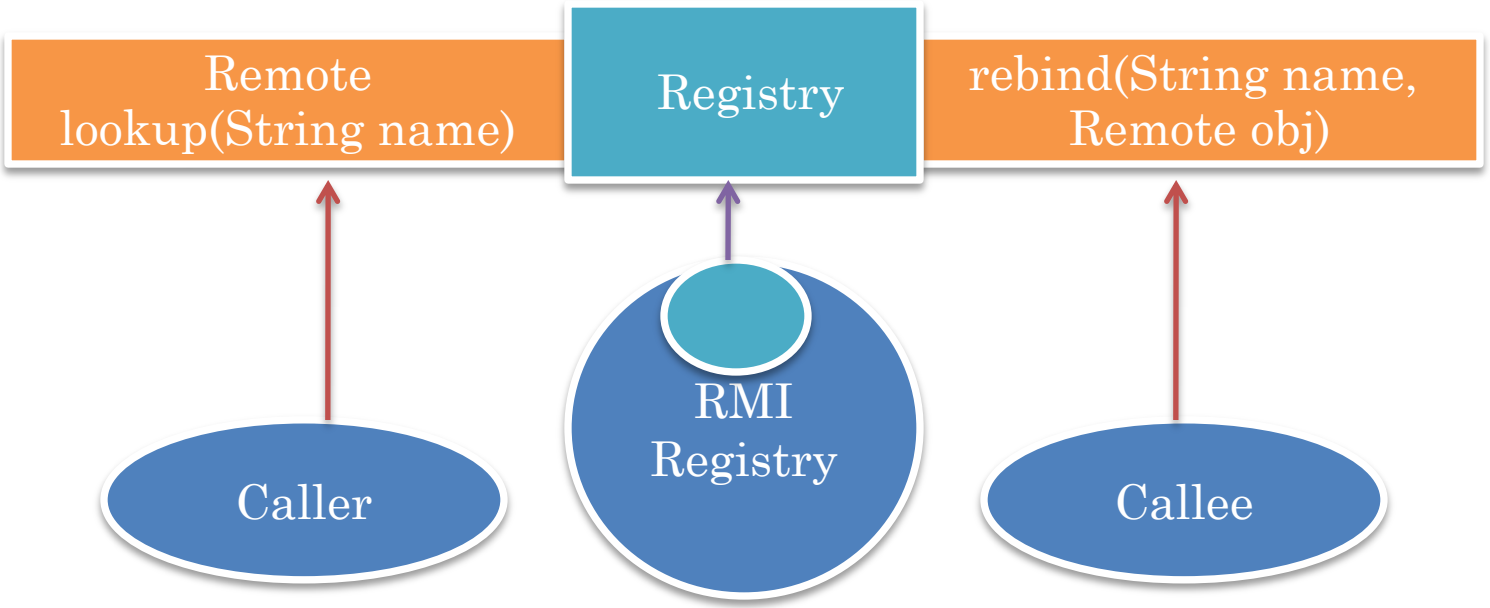
# Starting RMI Server From a Program

```
public class RMIRegistryStarter {
  public static void main (String[] args) {
    try {
      LocateRegistry.createRegistry(1099);
      Scanner scanner = new Scanner(System.in);
      scanner.nextLine();
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Prevents termination

Usually RMIRegistry started though LocateRegistry is part of server process

# Name Server

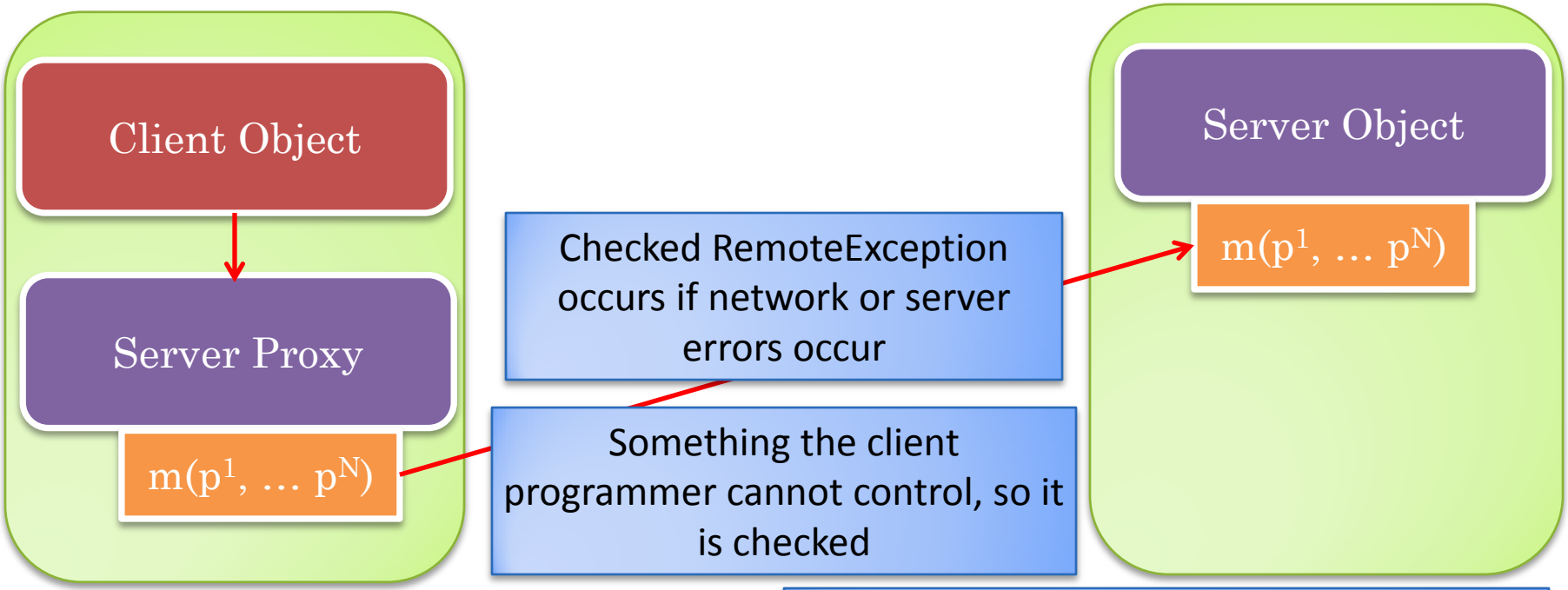



RMI Registry simply stores what was sent to it by rebind

How to create and store references rather than copies?

If a stub has been created, then instance of stub is sent as reference

# Proxies (Review)




Method call must catch RemoteException

Method declaration must indicate it may throw RemoteException in header

The class of server object must implement Remote

When is proxy class created? When is proxy created?

# Generating Proxy Class: Compilation

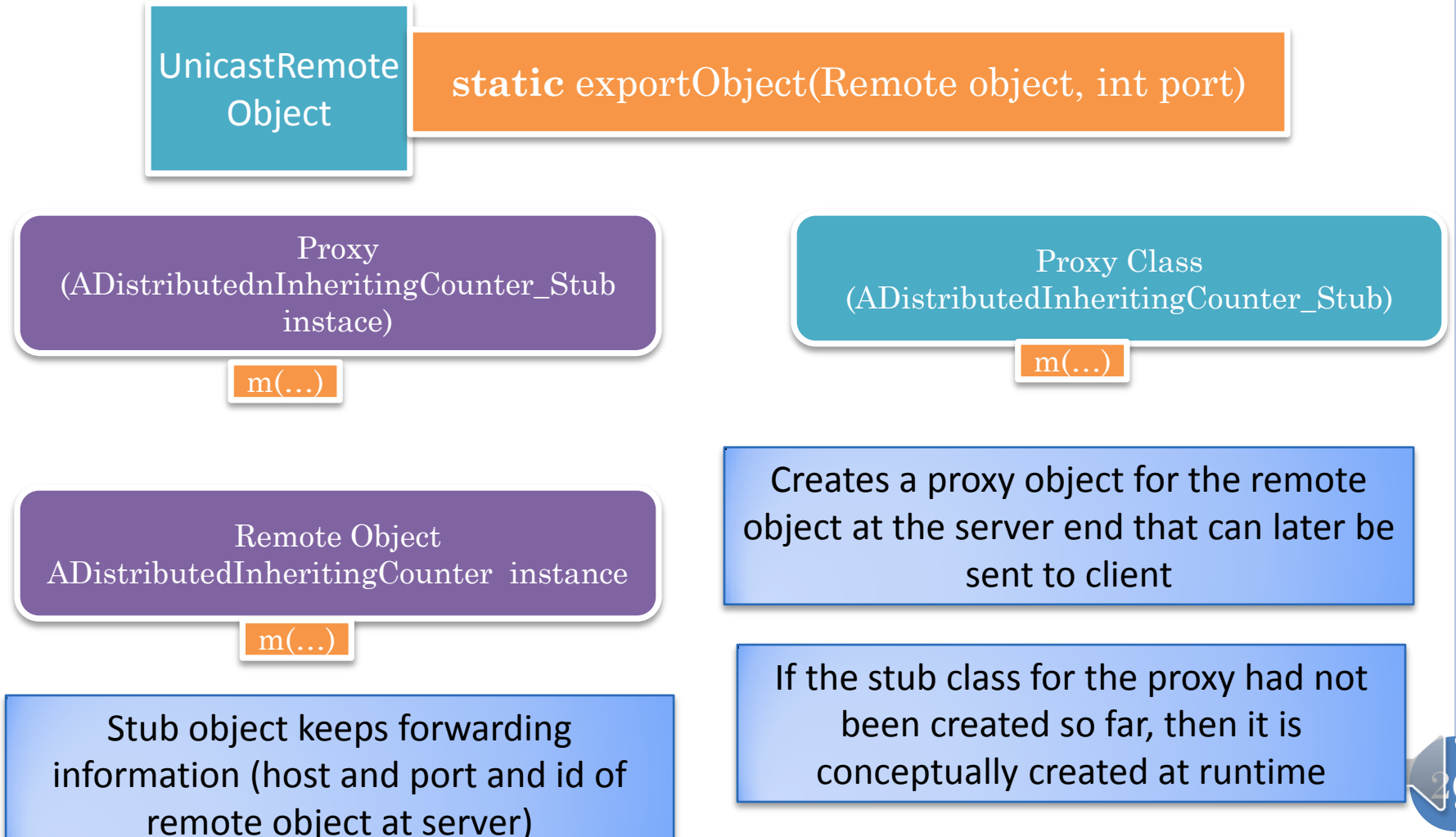
```
set javabin=D:\"Program Files"\Java\jdk1.6.0_03\bin
cd D:/dewan_backup/java/distTeaching/bin
%javabin%\rmic  rmi.examples. ADistributedInheritingRMICounter
```

```
Directory of D:\dewan_backup\Java\distTeaching\bin\rmi\examples
11/20/2011  09:12 AM    <DIR>          .
11/20/2011  09:12 AM    <DIR>          ..
11/19/2011  08:17 PM               933  ADistributedInheritingRMICounter.class
11/20/2011  09:12 AM             1,977  ADistributedInheritingRMICounter_Stub.class
11/20/2011  09:13 AM               264  DistributedRMICounter.class
11/19/2011  07:35 PM             1,112  DistributedRMICounterClient.class
11/19/2011  06:17 PM             1,154  DistributedRMICounterServer.class
11/19/2011  08:14 PM               908  RMIRegistryStarter.class
               6 File(s)          6,348 bytes
               2 Dir(s)  125,598,871,552 bytes free
```

Pre-compiler works from object code and produces object stub code

Eclipse will delete object code it has not generated

# Interpretive Reflection-based Class and Proxy Creation



# Invalid Remote Interface

```
public interface DistributedCounter extends Remote,
Serializable {
  void increment(int val) throws RemoteException;
  int getValue() throws RemoteException;
}
```

Interpretation

java.rmi.server.ExportException: remote object implements illegal remote interface; nested exception is:
java.lang.IllegalArgumentException: illegal remote method encountered:
public abstract void rmi.examples.DistributedCounter.increment(int)
at sun.rmi.server.UnicastServerRef.exportObject(Unknown Source)

Compilation

```
D:\dewan_backup\Java\distTeaching\bin>%javabin%\rmic
rmi.examples.AnInheritingDistributedRMICounter
error: rmi.examples.DistributedCounter is not a valid remote interface: method void increme
nt(int) must throw java.rmi.RemoteException.
1 error
```

# Local Method Parameter Passing

Either a copy or reference to a parameter is passed determined by whether it is  a call-by-value or call-by-reference

Caller and caller can share memory if  language has call-by-reference or pointers

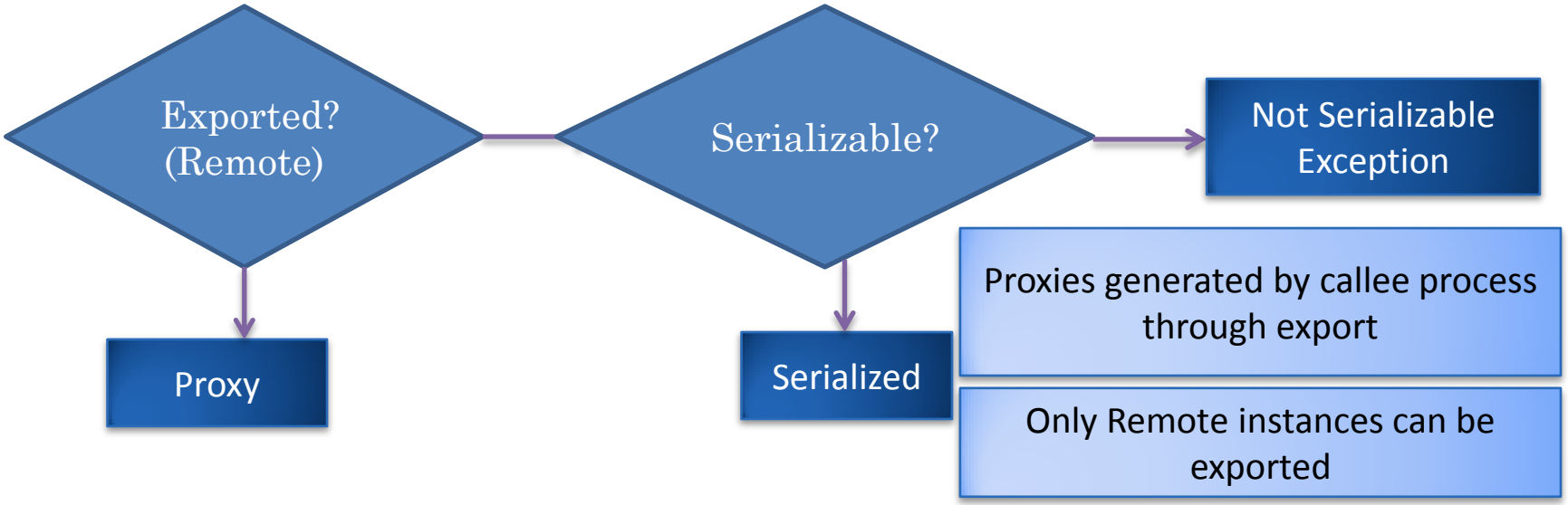
# Remote Method Parameter Passing

Either a serialized copy or proxy to a parameter is passed determined by … ?

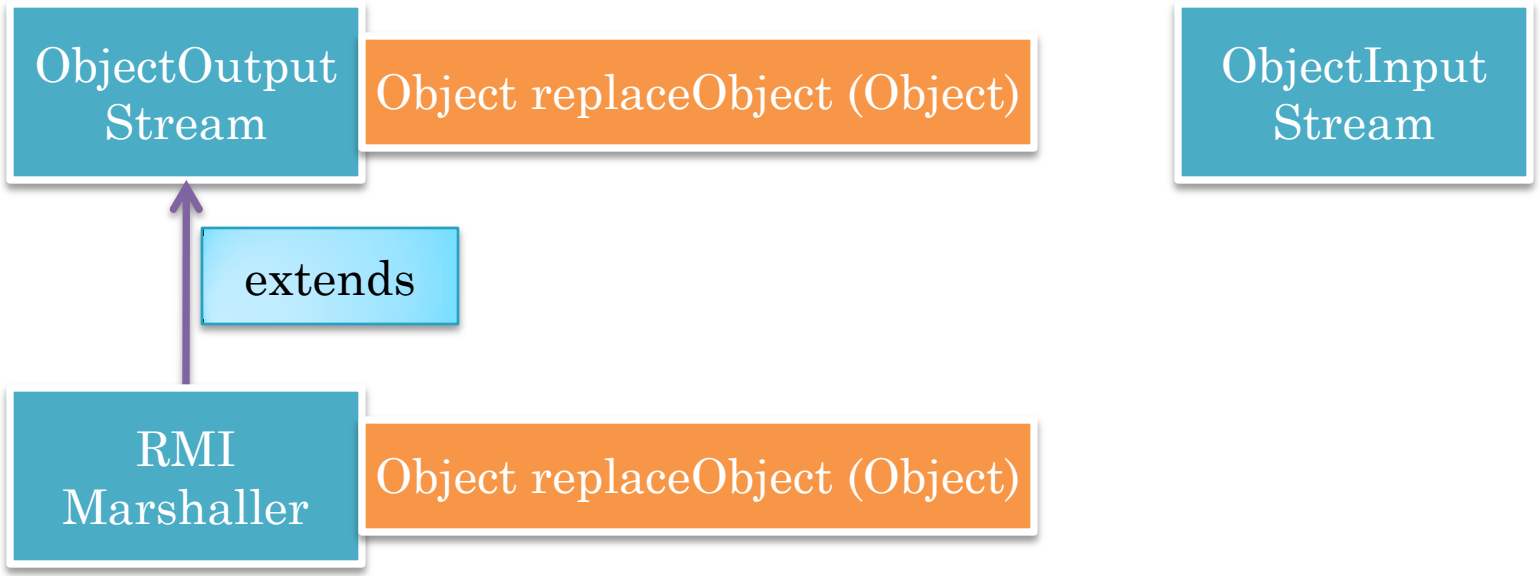
Caller and caller cannot share memory, so an address cannot be passed

# Copy or Proxy?

Should a parameter to a remote method be passed as a proxy or a serialized copy?




# Serialization →Marshalling



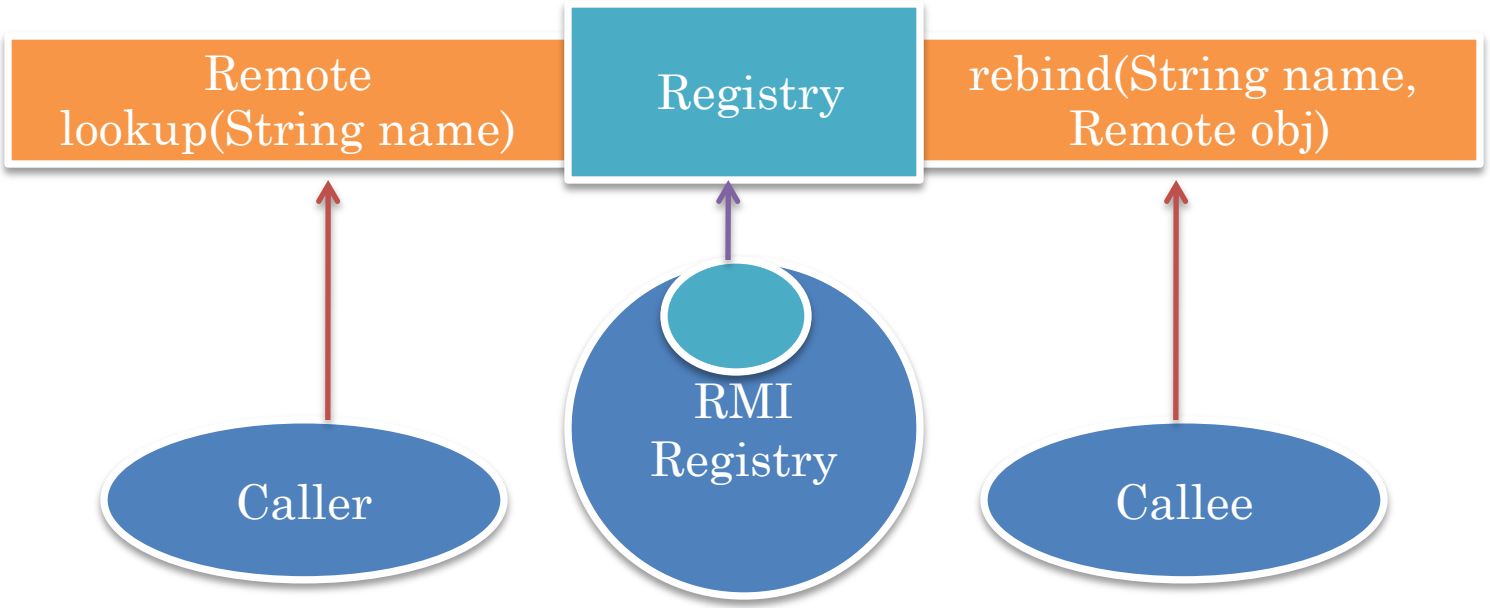

ObjectOutputStream calls replaceObject(object) to determine what object is actually serialized and sent

RMI Marshaller returns stub if object IS-A Remote and has been exported (at compile or runtime)

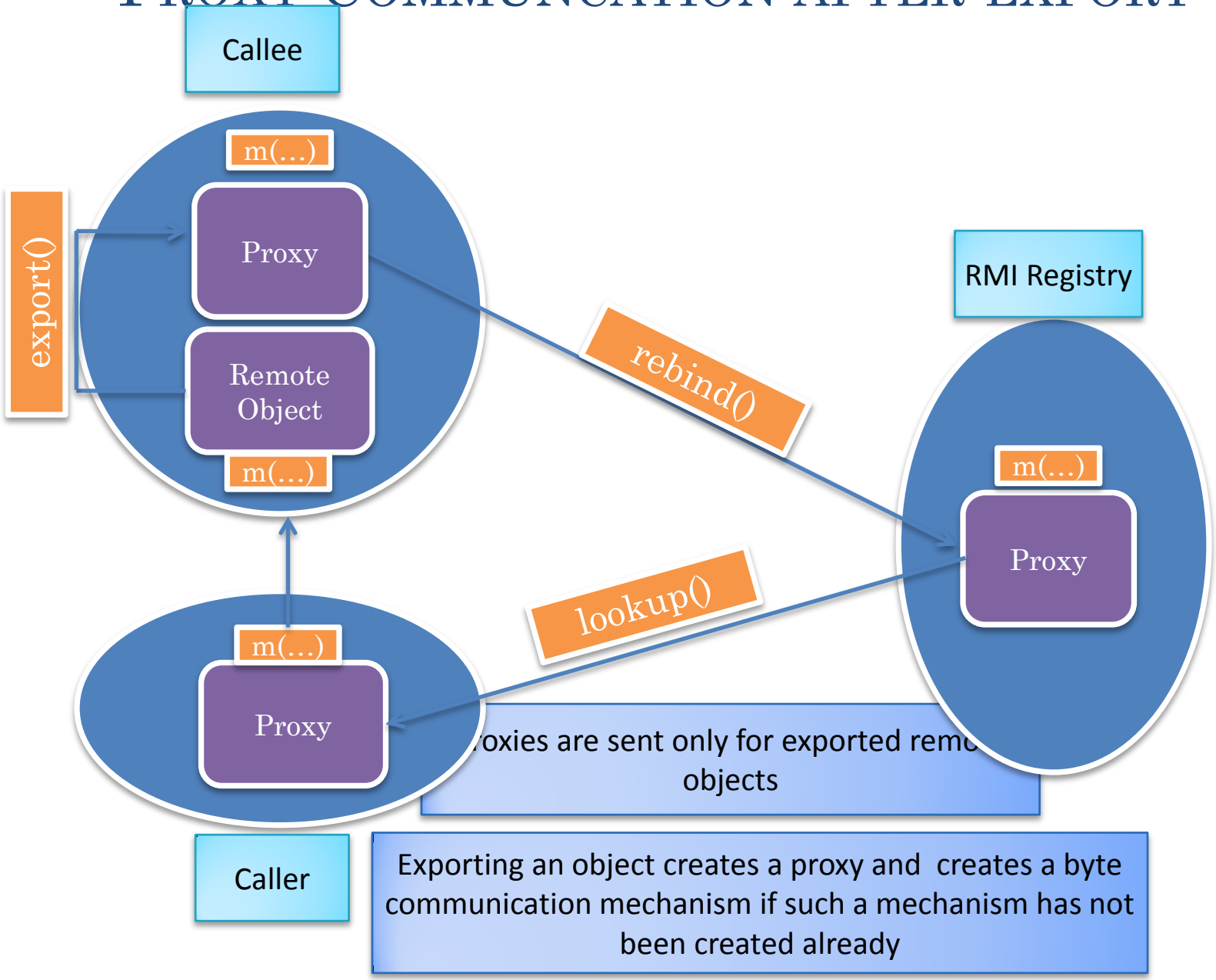
ObjectInputStream uses stub or copy

Marshaller and Serializer are tied to each other through inheritance

# Name Server (Review)



# Proxy Communcation after export



# Starting RMI Server From a Program (Review)

```
public class RMIRegistryStarter {
  public static void main (String[] args) {
    try {
      LocateRegistry.createRegistry(1099);
      Scanner scanner = new Scanner(System.in);
      scanner.nextLine();
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Prevents termination

Usually RMIRegistry started though LocateRegistry is part of server process

# Server Launcher && Export Object

```
public class CounterServer {
  public static void main (String[] args) {
    try {
      Registry rmiRegistry = LocateRegistry.getRegistry();
      DistributedCounter counter = new
                ADistributedInheritingRMICounter ();
      UnicastRemoteObject.exportObject(counter, 0);
      rmiRegistry.rebind(DistributedCounter.class.getName(),
counter);
      counter.increment(50);
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Use default port

Increment action will not be seen

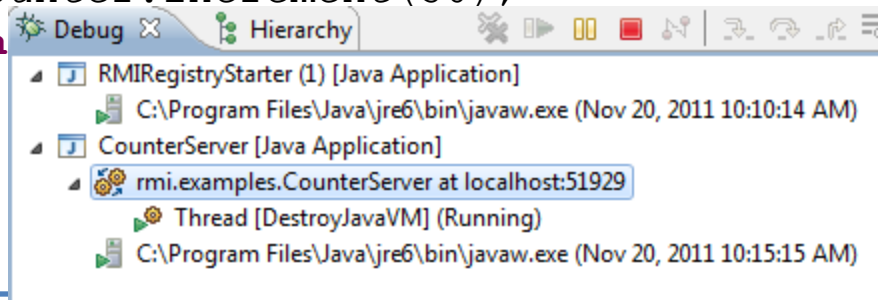
Serialized copy will be sent if object is serializable

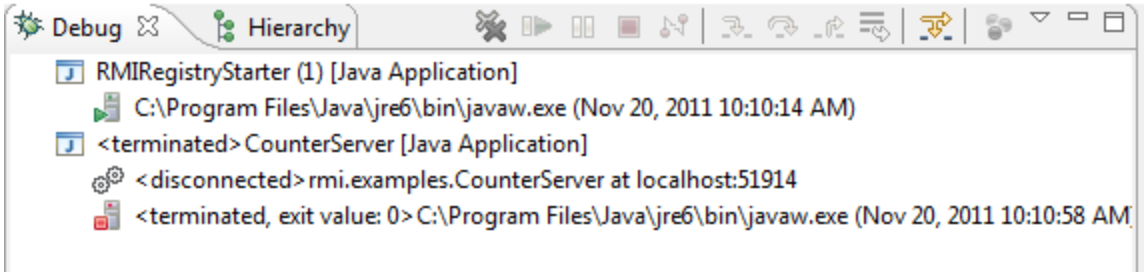
When does server terminate?

# Persistent Thread in Server

```
public class CounterServer {
  public static void main (String[] args) {
    try {
      Registry rmiRegistry = LocateRegistry.getRegistry();
      DistributedCounter counter = new
                ADistributedInteractingRMICounter ();
      UnicastRemoteObject.exportObject(counter, 0);
      rmiRegistry.rebind(DistributedCounter.class.getName(),
counter);
      counter.increment(50);
    } ca
      e.
    }
  }
}
```

Does not terminate



# Counter Client

```
public class CounterClient {
  public static void main (String[] args) {
    try {
    Registry rmiRegistry = LocateRegistry.getRegistry();
    DistributedCounter counter = (DistributedCounter )
        rmiRegistry.lookup(DistributedCounter.class.getName());
    System.out.println(counter.getValue());
} catch (Exception e) {
    e. printStackTrace();
}
```

Two different ways to get references, one for bootstrapping

# Understanding Equals (SERVER)

```
public class ServerRMICounterComparer extends RMICounterLauncher {
  public static void main (String[] args) {
    try {
      Registry rmiRegistry = LocateRegistry.getRegistry();
      DistributedRMICounter counter1 =
        new ADistributedInheritingRMICounter();
      DistributedRMICounter counter2 =
        new ADistributedInheritingRMICounter();
      UnicastRemoteObject.exportObject(counter1, 0);
      UnicastRemoteObject.exportObject(counter2, 0);
      rmiRegistry.rebind(COUNTER1, counter1);
      rmiRegistry.rebind(COUNTER2, counter2);
      DistributedRMICounter proxy1 =
        (DistributedRMICounter) rmiRegistry.lookup(COUNTER1);
      System.out.println(counter1.equals(counter2));
      System.out.println(counter1.equals(proxy1));
      System.out.println(counter1.hashCode() == proxy1.hashCode());
      System.out.println(proxy1);
    } ca
      e.
    }
  }
}
```

```
ServerRMICounterTester [Java Application] D:\Program Files\Java\jre1.6.0_04\bin\javaw.exe (Sep 27, 20
true
true
false
Proxy[DistributedRMICounter,RemoteObjectInvocationHandler[UnicastRef [liveRe
```

# Understanding Equals (SERVER) (Review)

```
public class ServerRMICounterComparer extends RMICounterLauncher {
  public static void main (String[] args) {
    try {
      Registry rmiRegistry = LocateRegistry.getRegistry();
      DistributedRMICounter counter1 =
        new ADistributedInheritingRMICounter();
      DistributedRMICounter counter2 =
        new ADistributedInheritingRMICounter();
      UnicastRemoteObject.exportObject(counter1, 0);
      UnicastRemoteObject.exportObject(counter2, 0);
      rmiRegistry.rebind(COUNTER1, counter1);
      rmiRegistry.rebind(COUNTER2, counter2);
      DistributedRMICounter proxy1 =
        (DistributedRMICounter) rmiRegistry.lookup(COUNTER1);
      System.out.println(counter1.equals(counter2));
      System.out.println(counter1.equals(proxy1));
      System.out.println(counter1.hashCode() == proxy1.hashCode());
      System.out.println(proxy1);
    } ca
      e.
    }
  }
}
```

```
ServerRMICounterTester [Java Application] D:\Program Files\Java\jre1.6.0_04\bin\javaw.exe (Sep 27, 20
true
true
false
Proxy[DistributedRMICounter,RemoteObjectInvocationHandler[UnicastRef [liveRe
```

# Understanding Equals (Client)

```
public class ClientRMICounterTester extends RMICounterLauncher {
  public static void main (String[] args) {
    try {
      Registry rmiRegistry = LocateRegistry.getRegistry();
      DistributedRMICounter counter11 =
         (DistributedRMICounter) rmiRegistry.lookup(COUNTER1);
      DistributedRMICounter counter12 =
         (DistributedRMICounter)    rmiRegistry.lookup(COUNTER1);
      DistributedRMICounter counter2 =
         (DistributedRMICounter) rmiRegistry.lookup(COUNTER2);
      System.out.println(counter12 == counter11);
      System.out.println(counter12.equals(counter11));
      System.out.println(counter11.hashCode() == counter12.hashCode());
      System.out.println(counter11.equals(counter2));
      System.out.println(counter11.hashCode() == counter2.hashCode());
      System.out.println(counter12);
    } catch (Exception e) {
      e.
    }
  }
}
```

```
<terminated> ClientRMICounterTester [Java Application] D:\Program Files\Jav
false
true
true
false
false
Proxy[DistributedRMICounter,RemoteObjectInvocationHandler[
```

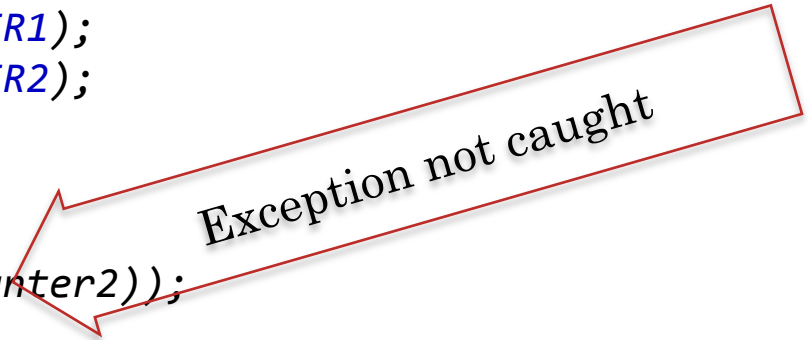
# Add Equals in Remote Interface?

```
public interface Counter {
  void increment(int val);
  Object getValue();
}
```

```
public interface DistributedRMICounter extends Remote {
  void increment(int val) throws RemoteException;
  Object getValue() throws RemoteException;
  boolean equals(Object otherObject) throws RemoteException;
}
```

# Add Equals in Remote Interface?

```
public static void remoteEqualsIssue () {
  Object counter1 = null;
  Object counter2 = null;
  try {
    Registry rmiRegistry = LocateRegistry.getRegistry();
    counter1 = rmiRegistry.lookup(COUNTER1);
    counter2 = rmiRegistry.lookup(COUNTER2);
  } catch (Exception e) {
    e.printStackTrace();
  }
  System.out.println(counter1.equals(counter2));
}
```




Conceptual issues arise because every object has equals

Remote IS-A Object

Interface IS-A Class!!

# RMI Limitation

Cannot call Object methods remotely

e.g. equals()

e.g. toString() (ObjectEditor uses it extensively)

# RMI Repository Shared by Multiple Clients

```
public class ARemoteRepository implements RemoteRepository {
  List<Remote> remotes = new ArrayList();
  public void deposit(Remote anObject) {
    remotes.add(anObject);
  }
  public List<Remote> getObjects() {
    return remotes;
  }
}
```

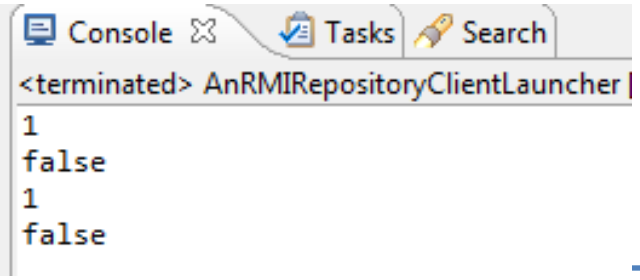
# RMI Server Holding Repository

```
public class AnRMIRepositoryServerLauncher
                  extends RemoteRepositoryLauncher {
  public static void main (String[] args) {
    try {
      Registry rmiRegistry = LocateRegistry.getRegistry();
      RemoteRepository repository = new ARemoteRepository();
      UnicastRemoteObject.exportObject(repository, 0);
      rmiRegistry.rebind(COUNTER_REPOSITORY, repository);
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

# Sharing Client

```
public class AnRMIRepositoryClientLauncher
                extends RemoteRepositoryLauncher{
  public static void main (String[] args) {
    try {
      Registry rmiRegistry = LocateRegistry.getRegistry();
      RemoteRepository counterRepository = (RemoteRepository)
             rmiRegistry.lookup(COUNTER_REPOSITORY);
      DistributedRMICounter exportedCounter = new
          ADistributedInheritingRMICounter();
      UnicastRemoteObject.exportObject(exportedCounter, 0);
      counterRepository.deposit(exportedCounter);
      exportedCounter.increment(1);
      List<Remote> objects = counterRepository.getObjects();
      for (Remote counter:objects) {
        System.out.println(((DistributedRMICounter) counter).getValue());
        System.out.println(counter == exportedCounter);
      }
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

```
AnRMIRepositoryClientLauncher
1
false
```



# INHERITING DISTRIBUTEDRMICOUNTER

```
public class ADistributedInheritingRMICounter extends ACounter
implements DistributedRMICounter{
  @Override
  public boolean equals(Object otherObject) {
    if (!(otherObject instanceof DistributedRMICounter))
      return super.equals(otherObject);
    try {
      return getValue().equals(
             ((DistributedRMICounter) otherObject).getValue());
    } catch (RemoteException e) {
      e.printStackTrace();
      return false;
    }
  }
}
```

IS-A link statically bound to reused code but requires less work

# Delegation to Counter

```
public class ADistributedDelegatingRMICounter
        extends UnicastRemoteObject implements DistributedRMICounter{
    Counter counter = new ACounter();
    public ADistributedDelegatingRMICounter() throws RemoteException {
      super();
    }
    public Object getValue() {
      return counter.getValue();
    }
    public void increment(int val) {
      counter.increment(val);
    }
     public boolean equals(Object otherObject) {
      if (!(otherObject instanceof DistributedRMICounter))
        super.equals(otherObject);
      try {
         return getValue().equals(
               ((DistributedRMICounter) otherObject).getValue());
      } catch (RemoteException e) {
        e.printStackTrace();
        return f
      }
    }
}
```

Super constructor automatically exports object

Correct semantics of hashcode

HAS-A link can be dynamically bound to reused code but requires more work

# Comparer

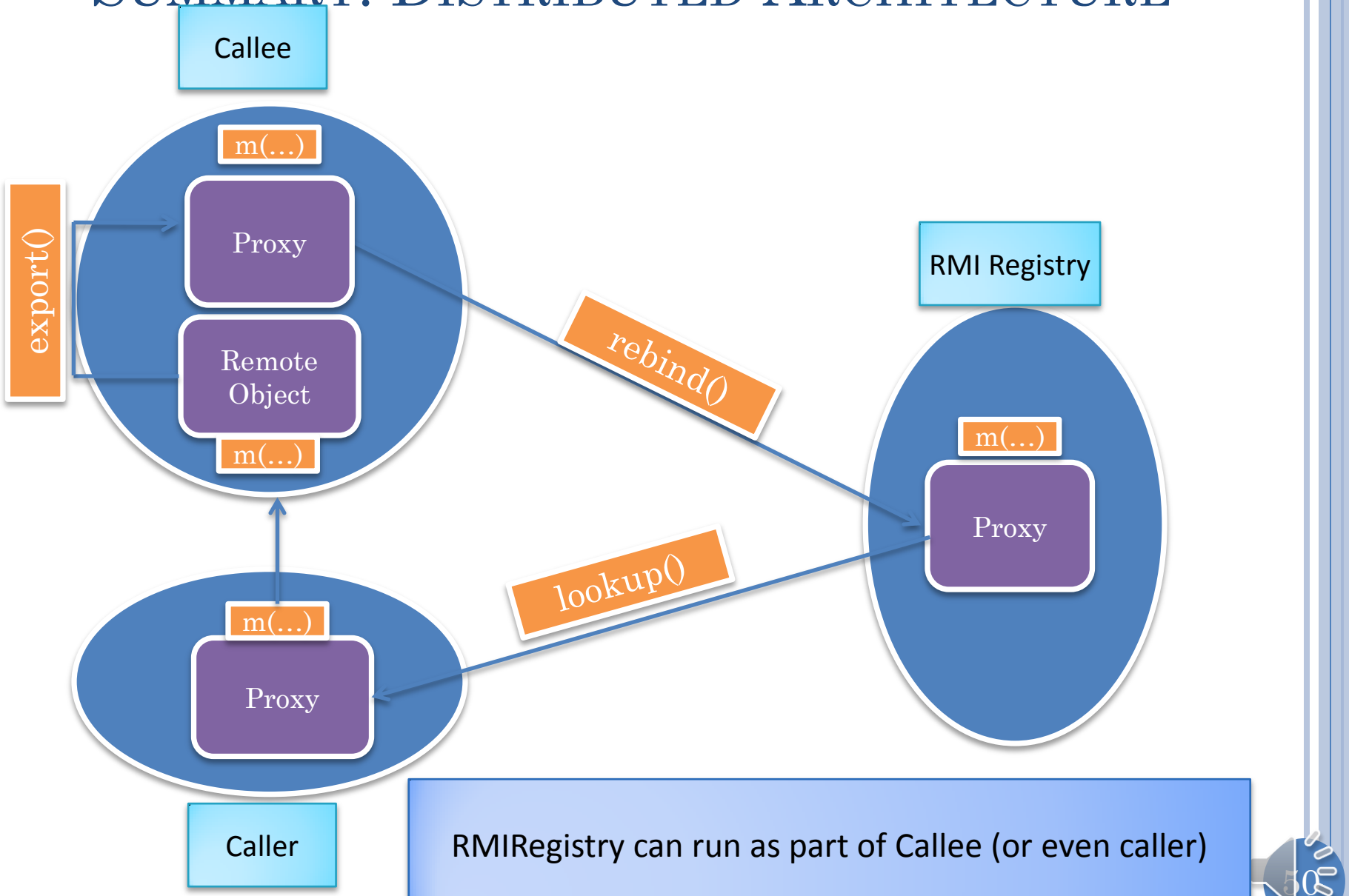
```
public class DelegatingServerRMICounterComparer
             extends RMICounterLauncher {
  public static void main (String[] args) {
    try {
      Registry rmiRegistry = LocateRegistry.getRegistry();
      DistributedRMICounter counter1 =
        new ADistributedDelegatingRMICounter();
      DistributedRMICounter counter2 =
        new ADistributedDelegatingRMICounter();
      rmiRegistry.rebind(COUNTER1, counter1);
      rmiRegistry.rebind(COUNTER2, counter2);
      DistributedRMICounter proxy1 =
        (DistributedRMICounter)    rmiRegistry.lookup(COUNTER1);
      System.out.println(counter1.equals(counter2));
      System.out.println(counter1.equals(proxy1));
      System.out.println(counter1.hashCode() == proxy1.hashCode());
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

No exportObject()

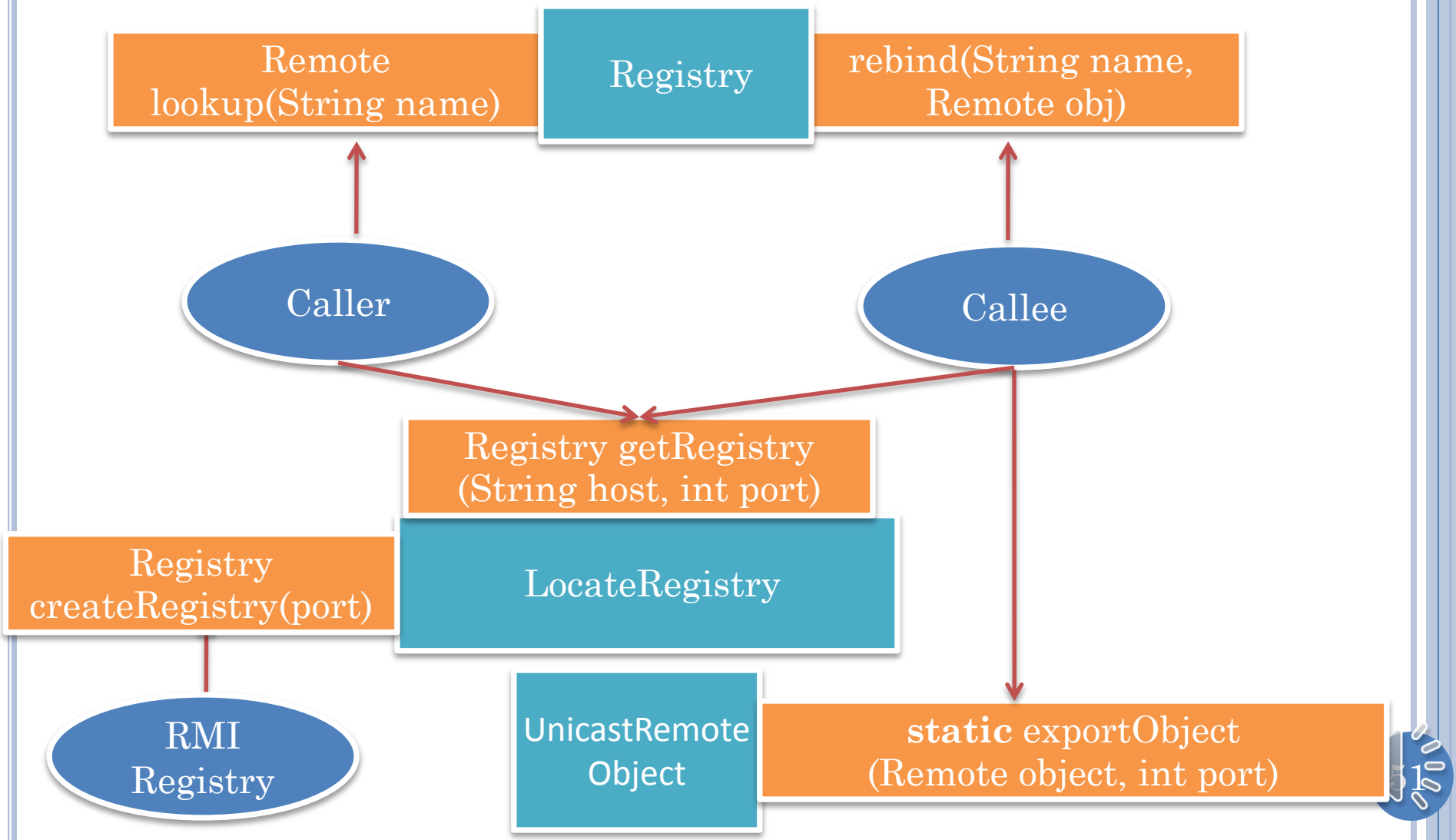
DelegatingServerRMICounterComparer

```
true
true
true
```

# Summary: Distributed Architecture



# Summary: RMI API



# Summary

- Java RMI is built on a data communication layer
- To allow methods of an object to be invoked remotely, the process containing an object must first generate a proxy for it and then register the proxy in an RMI registry
  - Such a process does not terminate when its main method exits.
- The process wishing to invoke methods on a remote object must fetch a serialized version of its proxy from the registry, which contains proxy methods for the remote methods of the remote object.
- Both a client and a server can register and lookup objects.
- The lookup must be done after the register.
- This means that a server must wait for some call from the client before looking up a proxy for it.
  - A remote method call can ask RMI for the host of the caller to generate the proxy
  - Usually the client will send a proxy with a call as we will see later, but we have illustrated the getHost() call

# Summary

- Proxy methods marshal method calls into messages on the data communication and unmarshal received message into return values.
- Errors on the communication channel are not under the control of the programmer, so remotely invoked methods must acknowledge the checked RemoteException
  - In a proxy for a remote object, proxy methods are generated only for remote methods of the object
  - All methods of a class implementing and an interface extending Remote are remote methods.
  - A remote method must acknowledge the checked RemoteException
  - This checking is done when the proxy is generated.

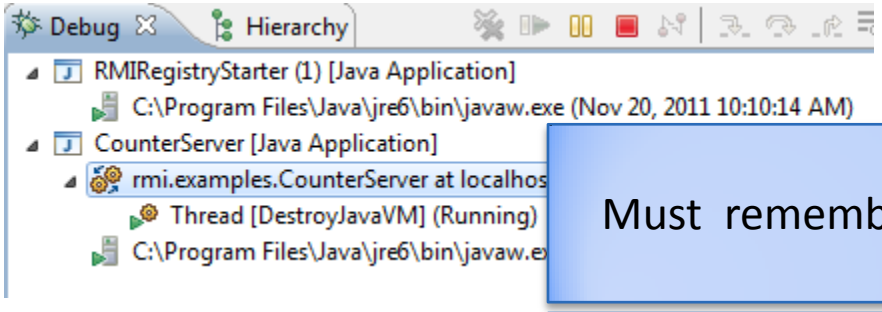
# Summary

- This means Object methods such as equals(), toString() and hashcode() cannot be invoked remotely
- It also means that if we want to create a central model connected to distributed views/controllers, then all of these classes must be made distribution aware.
  - Can reuse existing non distribution unaware versions of these classes by inheriting from them or delegating to them.

# Summary

- In RMI, each remote object associated with a server communication channel, that is, a channel to which any process can connect.
- As a result a proxy to a remote object can be sent to any other process.
- In RMI a parameter to a remote method is sent by reference (as a proxy) if it is of type Remote and by copy if it is Serializable.
- In RMI, two proxies to the same remote object are not == to each other
- If an exported object is fetched, a proxy rather than the original object is returned, but the equal() method is locally called correctly.

# Launching Multiple Processes on One Host in Eclipse




Must remember which programs are to be launched

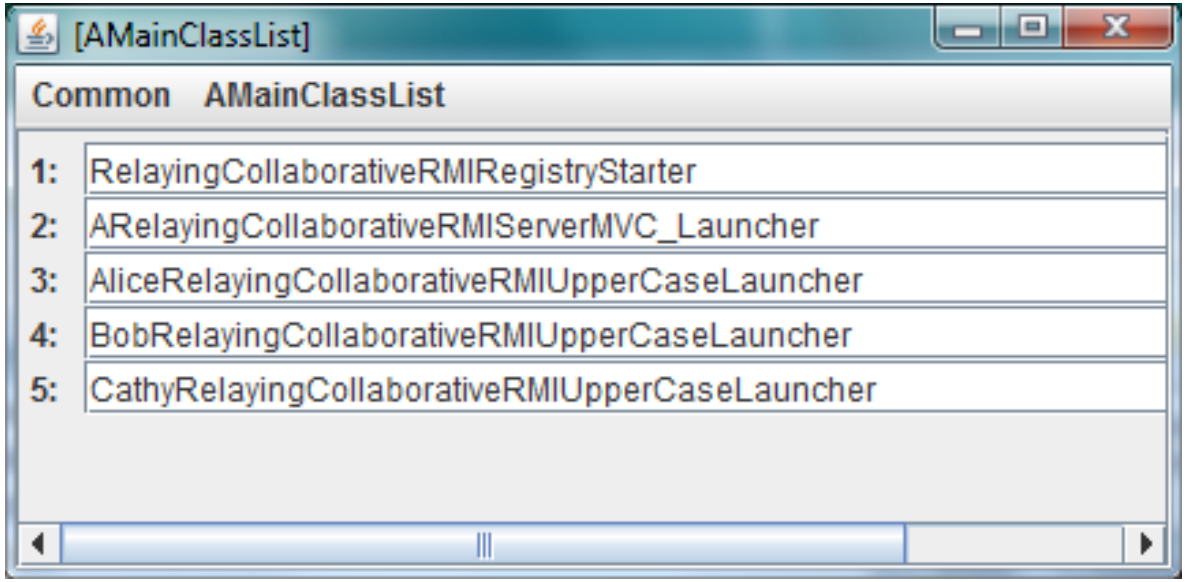
Must remember the order (.eg. Registry before Server before client)

Can see the console I/ of only one process at one time

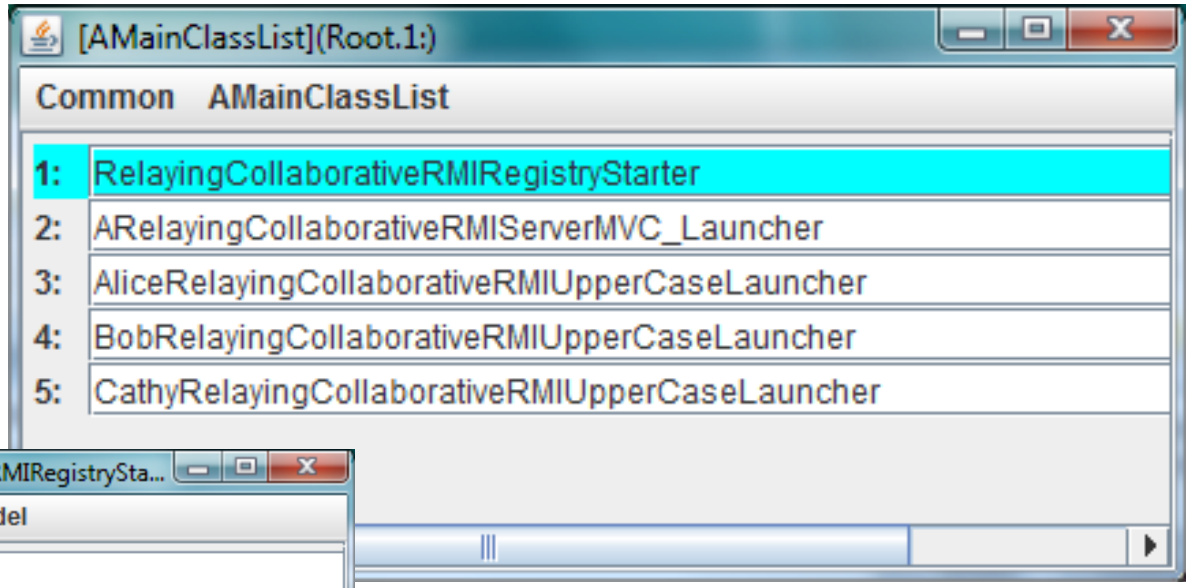
Must manually start and kill each process and select the console-window process

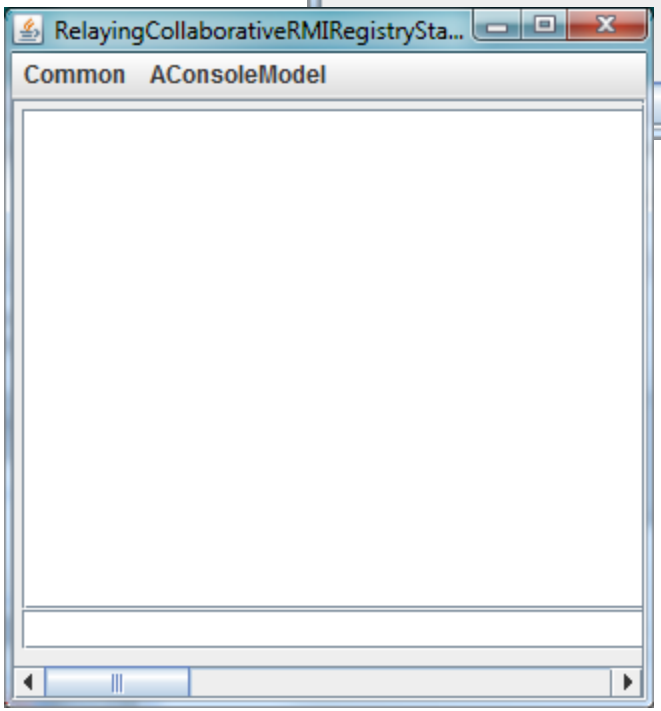
Running from the command window is unfamiliar and requires swithcing away from Eclipse

# Launching Multiple Processes: UI

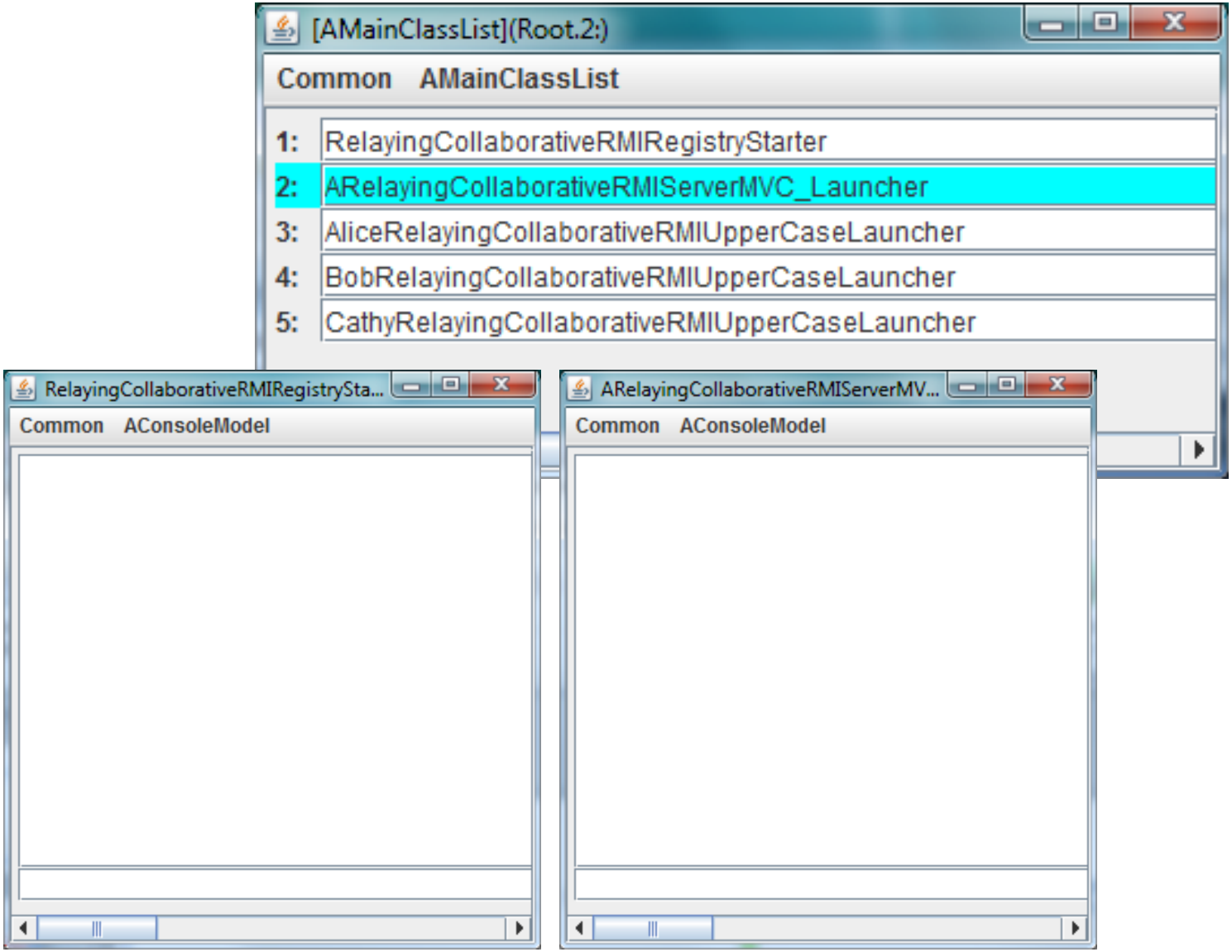


# Double clicking On Registry



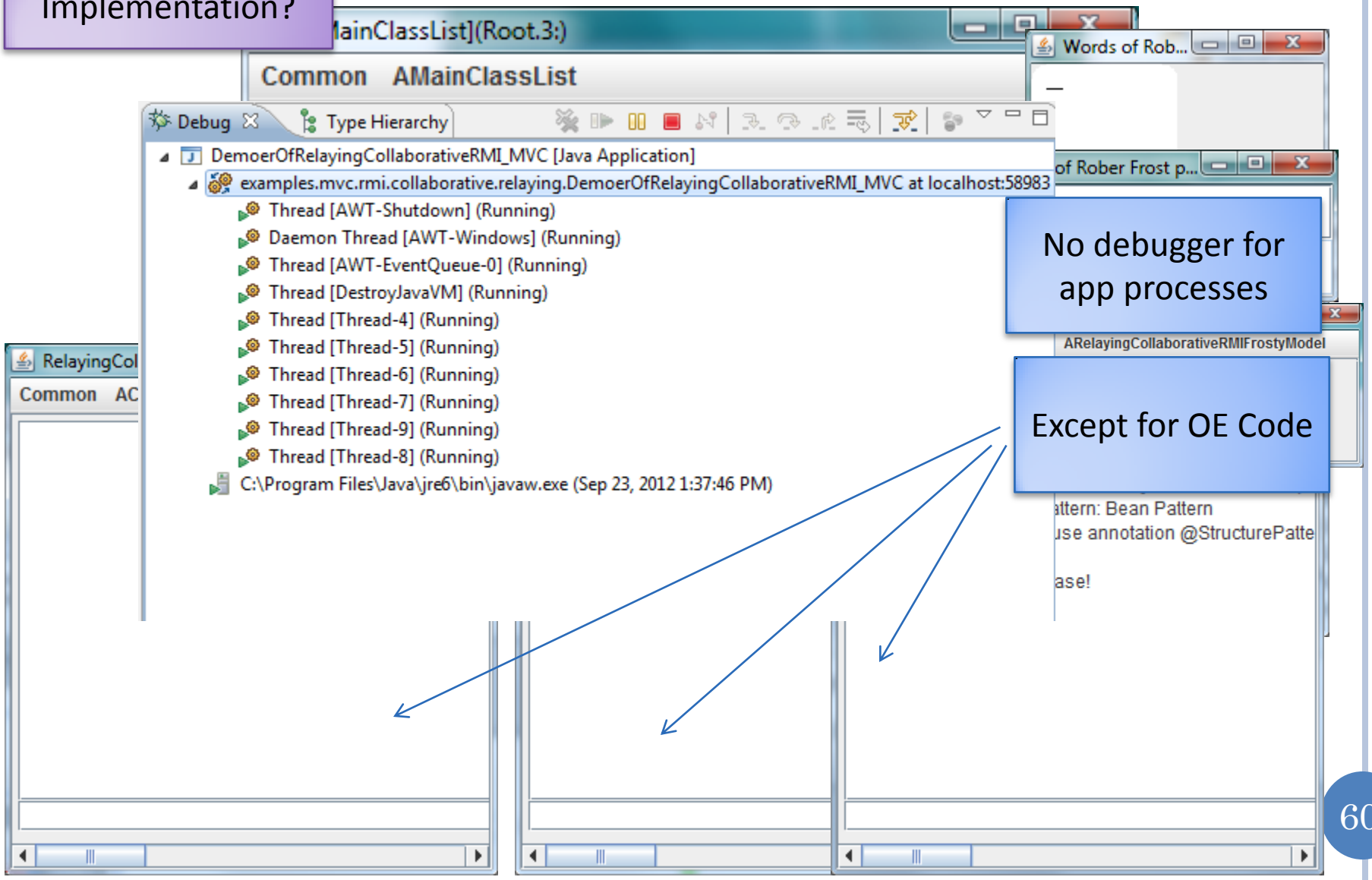



# Double clicking On Server

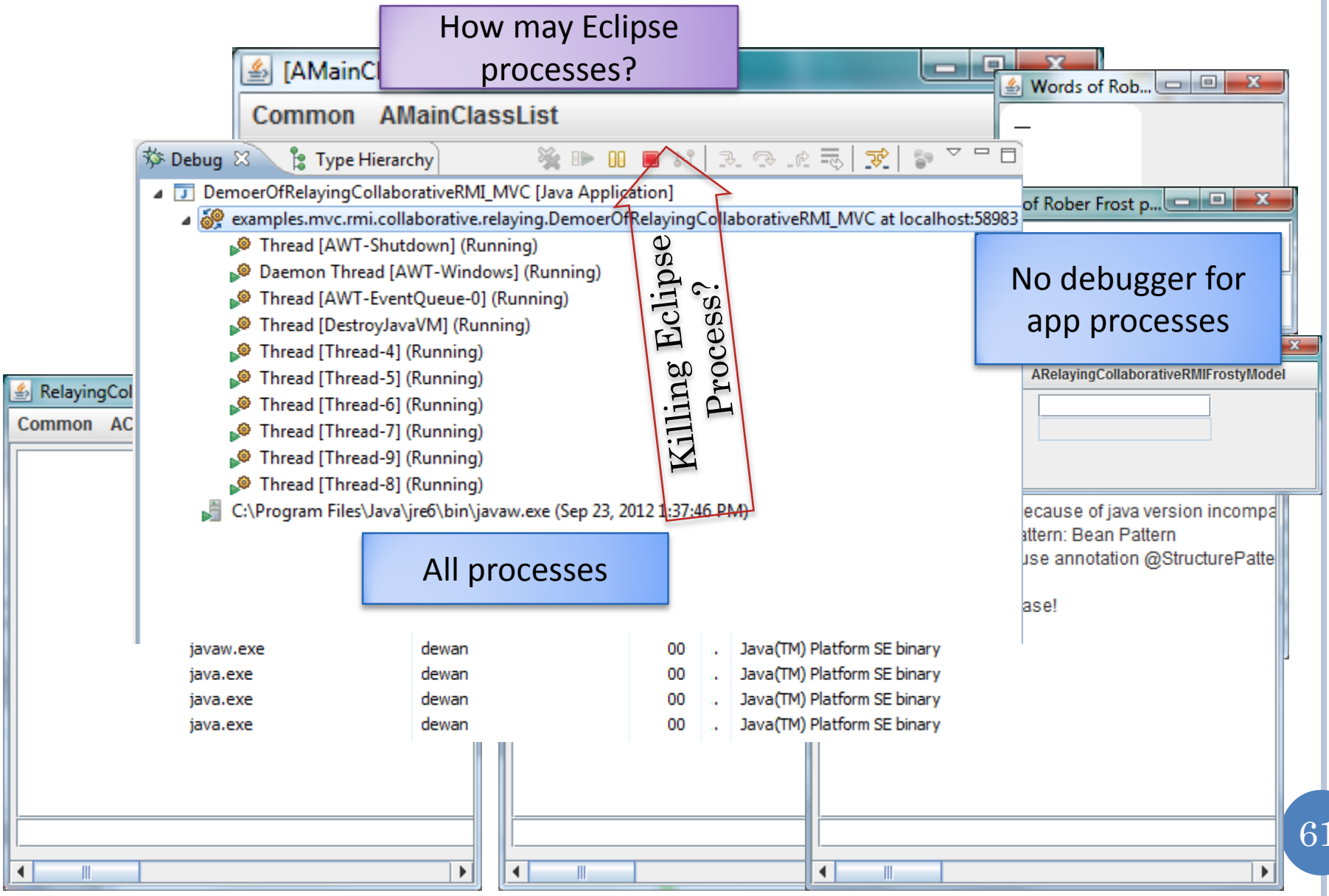


# Double clicking On Alice Launcher

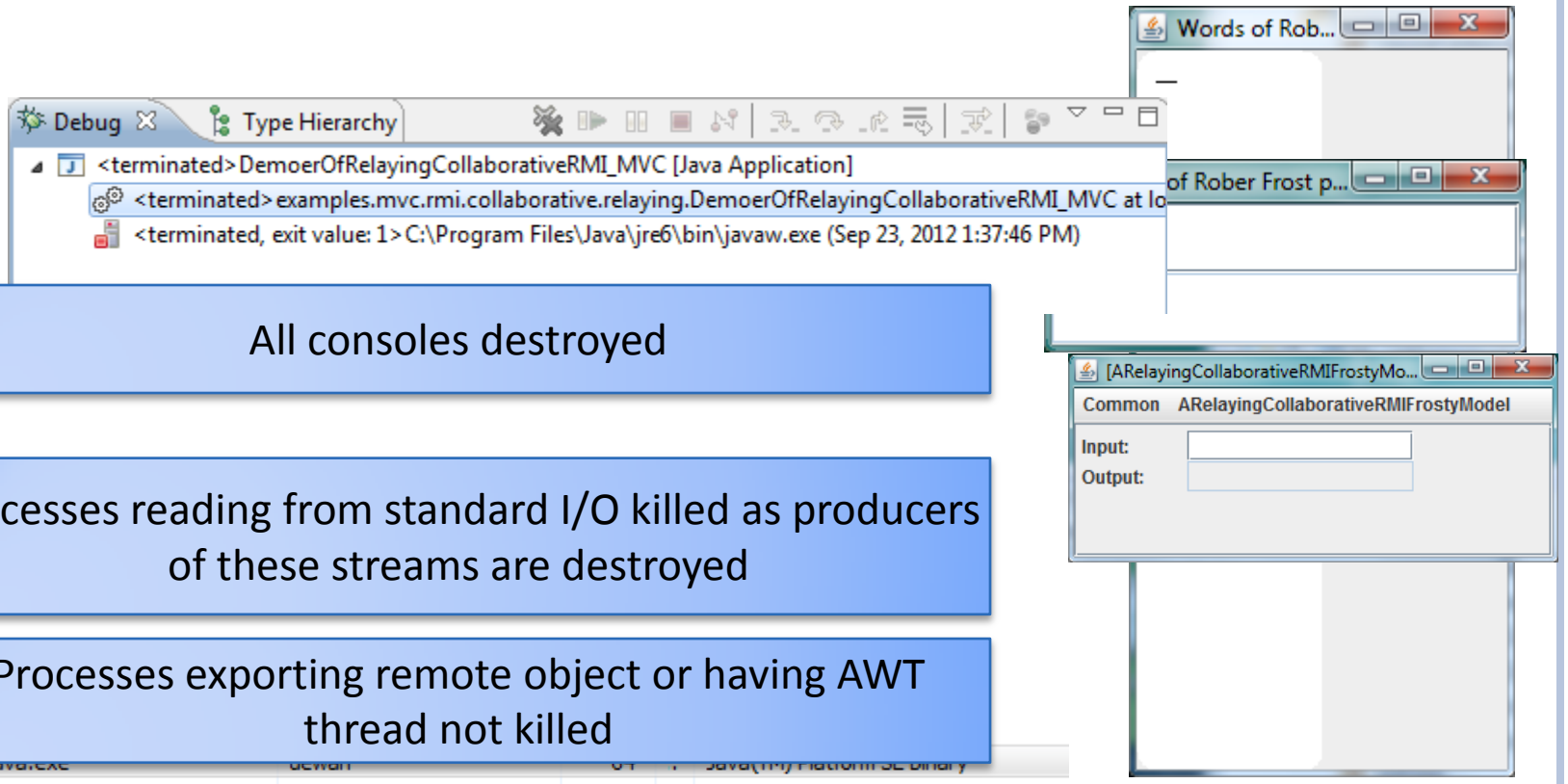
Implementation?



# Eclipse vs. Java Processes



# Killing Eclipse Process




All consoles destroyed

Processes reading from standard I/O killed as producers of these streams are destroyed

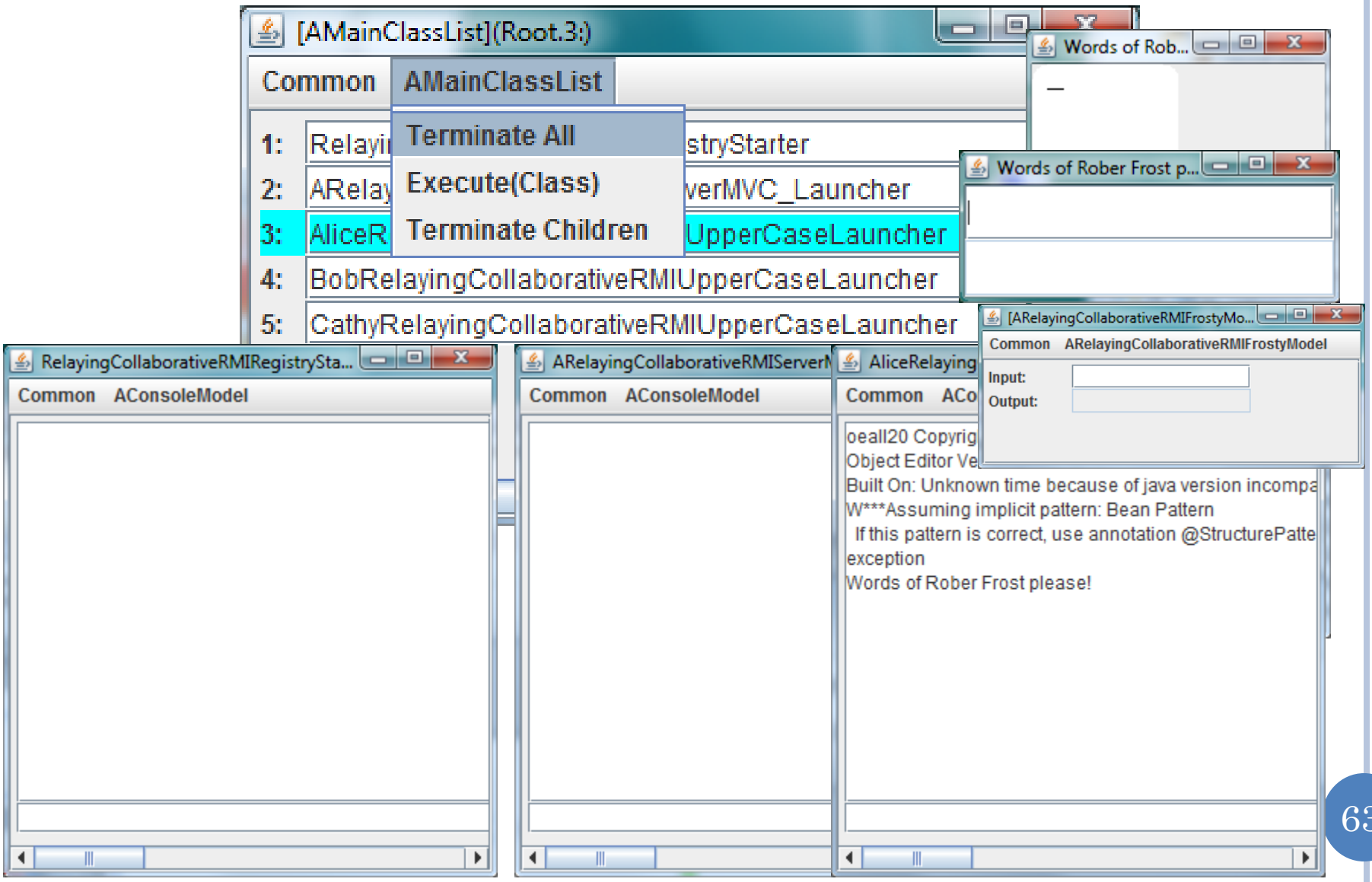
Processes exporting remote object or having AWT thread not killed

Could eclipse process not detect its killing an destroy its forked processes?

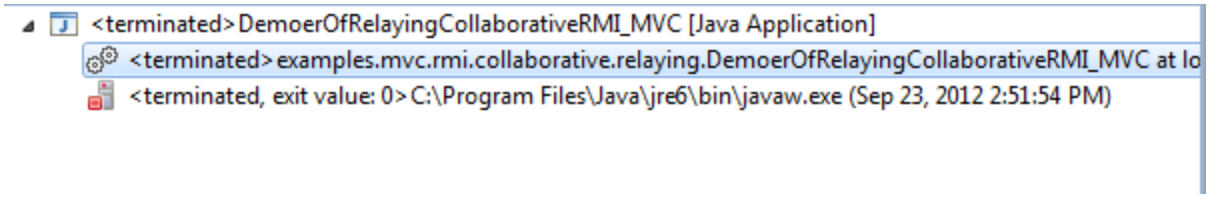
Regular but not eclipse process cannot

Eclipse calls Process.destroy() and not OS kill

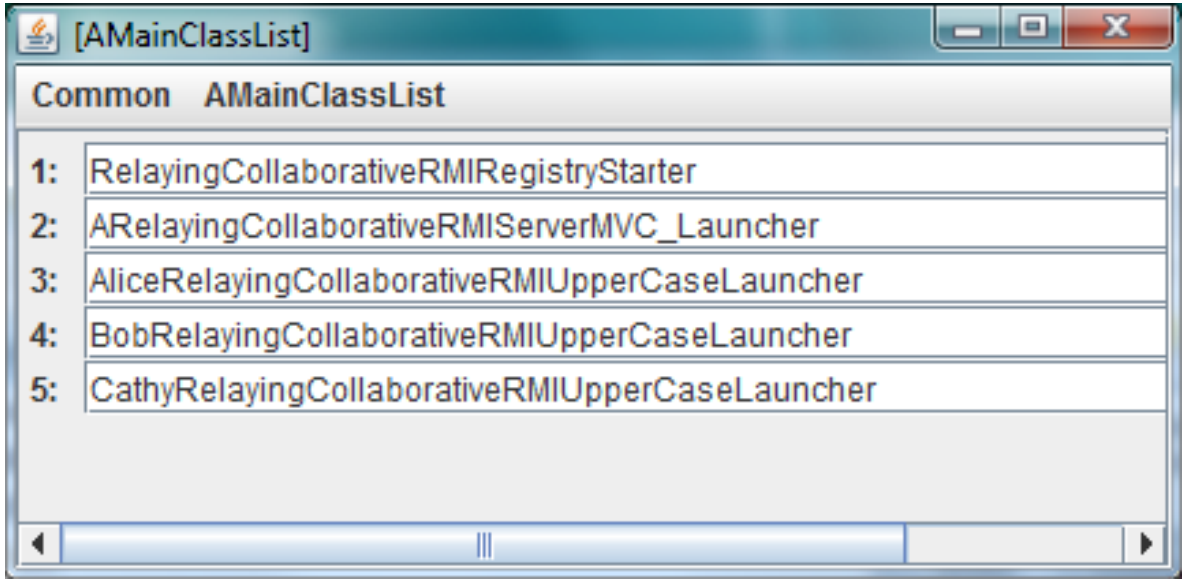
# KILLING FROM AMAINCLASSLIST



# All Processes Killed



# Local and Remote Response




Programming Interface?

# Multi-Process Launcher API

```
import bus.uigen.models.MainClassListLauncher;
public class DemoerOfRelayingCollaborativeRMI_MVC {
  public static void main(String args[]) {
    demo();
  }
  public static void demo() {
    Class[] classes = {
      RelayingCollaborativeRMIRegistryStarter.class,
      ARelayingCollaborativeRMIServerMVC_Launcher.class,
      AliceRelayingCollaborativeRMIUpperCaseLauncher.class,
      BobRelayingCollaborativeRMIUpperCaseLauncher.class,
      CathyRelayingCollaborativeRMIUpperCaseLauncher.class
    };
    MainClassListLauncher.launch(classes);
 }
}
```

# Multi-Process Launcher

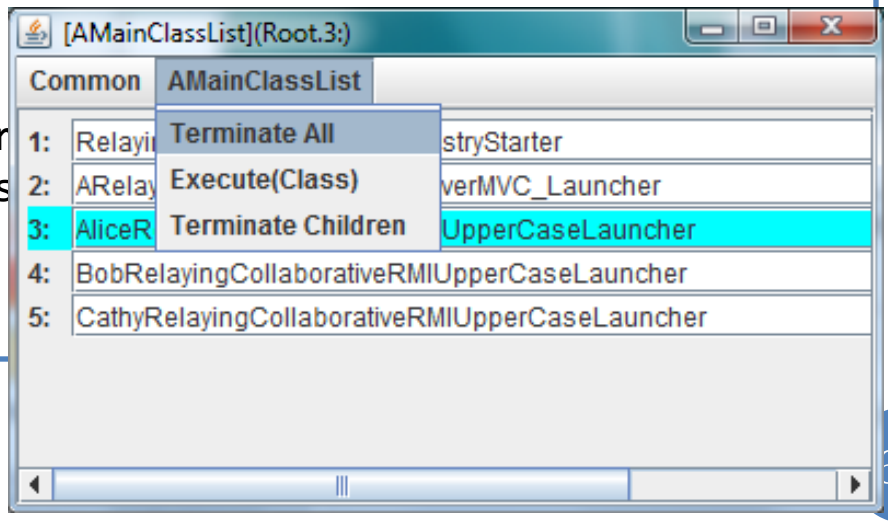
```
package bus.uigen.models;
import util.models.ListenableVector;
import bus.uigen.ObjectEditor;
public class MainClassListLauncher {
  public static void launch(Class[] classes) {
    ListenableVector<Class> classList = new AMainClassList();
    for (Class aClass:classes) {
      classList.add(aClass);
    }
    ObjectEditor.edit(classList);
  }
}
```

# Main Class List

```
import java.util.ArrayList;
import java.util.List;
import util.annotations.Visible;
import util.models.AListenableVector;
import util.remote.ProcessExecer;
import bus.uigen.misc.OEMisc;
public class AMainClassList extends AListenableVector<Class>
                implements Runnable {
  List<ProcessExecer> executed = new ArrayList();
  public AMainClassList() {
    Thread thread = new Thread(this);
    Runtime.getRuntime().addShutdownHook(thread);
  }
  @Visible(false)
  public void run() {
     killAllChildren();
  }
```

# Main Class List

```
  public void open(Class element) {
    executed.add(OEMisc.runWithObjectEditorConsole(element, ""));
  }
  public void execute(Class element) {
    open(element);
  }
  public void terminateChildren() {
    killAllChildren();
  }
  public void terminateAll() {
    System.exit(0);
  }
  void killAllChildren() {
    for (ProcessExecer processExecer
      processExecer.getProcess().des
    }
  }
}
```



# Instantiating and Editing Process Execer

```
public static ProcessExecer runWithObjectEditorConsole(
   Class aJavaClass, String args) {
  ProcessExecer processExecer =
     new AProcessExecer(aJavaClass, args);
  Process process = processExecer.execProcess();
  ConsoleModel consoleModel = processExecer.consoleModel();
  OEFrame frame = ObjectEditor.edit(consoleModel);
   consoleModel.initFrame(
      (Frame) frame.getFrame().getPhysicalComponent());
  frame.setTitle(consoleModel.getTitle());
  return processExecer;
}
```

# Process Execer

```
public class AProcessExecer implements ProcessExecer {
  Process process;
  String className;
  String args;
  ConsoleModel consoleModel;
  String command;
  String title;
  public AProcessExecer( Class aJavaClass, String anArgs) {
    className = aJavaClass.getName();
    args = anArgs;
    String classPath =
           System.getProperty("java.class.path");
    command = "java -cp " + classPath + " " + className + " " + args;
    title = aJavaClass.getSimpleName() + " " + args;
}
```

# Process Execer

```
public Process execProcess() {
  try {
    Runtime rt = Runtime.getRuntime();
    System.out.println("Execing command " + command);
    System.out.println("Working Directory = " +
    System.getProperty("user.dir"));
    File binDirectory = new File ("bin");
    process = rt.exec(command, null, binDirectory);
    consoleModel = new AConsoleModel(process, title);
  } catch (Exception e) {
    e.printStackTrace();
    return null;
  }
  return process;
 }
}
```

# Exporting State

```
  public Process getProcess() {
    return process;
  }
  public ConsoleModel consoleModel() {
    return consoleModel;
  }
  public void destroy() {
    process.destroy();
  }
  public String getTitle() {
    return title;
  }
}
```

# AConsoleModel

```
public class AConsoleModel implements ConsoleModel {
  String input = "";
  StringBuilder output = new StringBuilder("");
  Thread outputThread, errorThread;
  PrintStream printStream;
  Process process;
  String title;
  PropertyChangeSupport propertyChangeSupport;
  public AConsoleModel(Process aProcess, String aTitle) {
    propertyChangeSupport = new PropertyChangeSupport(this);
    process = aProcess;
    title = aTitle;
    printStream = new PrintStream(process.getOutputStream());
    outputThread = new Thread(new AConsoleModelStreamReader
       ("out", process.getInputStream(), this));
   errorThread = new Thread(new AConsoleModelStreamReader("error",
       process.getErrorStream(),  this));
   outputThread.start();
   errorThread.start();
  }
```

# AConsoleModel

```
  @ComponentWidth(1200)  @Position(1)
  public String getInput() {
    return input;
  }
  public void setInput(String newVal) {
    addOutput(newVal);
    printStream.println(newVal);
    printStream.flush();
    propertyChangeSupport.firePropertyChange(
       new PropertyChangeEvent(this, "input", null, input ));
  }
  @Visible(false)
  public void addOutput(String newVal) {
    output.append(newVal + "\n");
    propertyChangeSupport.firePropertyChange(
       new PropertyChangeEvent(this, "output", null, output ));
}
  @DisplayToString(true) @PreferredWidgetClass(JTextArea.class)
  @ComponentWidth(1200)  @Position(0)
  public StringBuilder getOutput() {
    return output;
  }
```

# AConsoleModel

```
  @Visible(false)
  public String getTitle() {
    return title;
  }
  public void exit() {
    process.destroy();
  }
  public void addPropertyChangeListener(PropertyChangeListener aListener) {
    propertyChangeSupport.addPropertyChangeListener(aListener);
  }
  @Visible(false)
  public void initFrame(Frame aFrame) { }
}
```

# Stream Reader

```
public class AConsoleModelStreamReader implements Runnable {
  BufferedReader bufferedReader;
  String type;
  ConsoleModel consoleModel;
  public AConsoleModelStreamReader(String aType,
       InputStream anInputStream, ConsoleModel aConsoleModel) {
    consoleModel = aConsoleModel;
    bufferedReader = new BufferedReader(
         new InputStreamReader(anInputStream));
  }
  public void run() {
    try {
      String line = null;
      while ((line = bufferedReader.readLine()) != null) {
        consoleModel.addOutput(line);
      }
    } catch (IOException ioe) {
      ioe.printStackTrace();
    }
   }
}
```

# Launcher

Need latest version of oeall22

Tracer.showWanrings(false) at start of main to suppress warnings from Beau's project, which does not conform to new version

# Summary

- Need to give local host as the location of all processes, no change needed in the rest of the application.
- Eclipse and other programming environments allow launching of only one process at a time and do not allow the console I/O of all processes to be viewed simultaneously
- We can use a script to run the processed in different command windows but then we must set the class path for launching programs from console windows.
- Java provides several for us to write a general mechanism to launch multiple processes from another process, set their paths, and redirect their I/O to our own simulations of command windows.
- The launching process can be a process launched by a programming environment such as Eclipse.

# Summary

- The Java Runtime.getRunTime().exec() call can be used to run (exec) the java command to create a non programming environment Java process represented by the Process object returned by the call.
  - This command can be given the class path of the programming environment process that creates the non programming environment processes
  - The class path of a process is in a property fetchable from the System class.
- A Process object has properties representing its input, output, and error streams, which can be used to create our own console for it.
  - Two threads can be created to read the output and error of the process and display it in our own console
  - Input trapped by our console can be written to the processes's input stream.
  - Counter-intuitevly a processes's input(output) stream is fetched by calling the getOutput(Input)Stream() method of the Process object representing it because these streams are output(input) streams for the process invoking methods on these objects.

# Summary

- When the launching process is killed we want to destroy all processes launched by it.
- A programming environment does not know about these processes, so it cannot kill them.
- So we must write our own code to kill these child processes.
- Java provides a mechanism for a process to detect when it is shutting down.
- However, this code is not called when a programming environment kills the process because it destroys the process rather than shuts it down.
- Shutting down can be done only by the OS
- The launching process can provide a user interface to manually call this code

# Summary

- We have used OE and these facilities to create a general abstraction for launching processes with separate consoles.
- The abstraction requires the programmer to specify a list of main classes to be executed and provides an API.
- The open method can then be called on this abstraction with one of these classes as an argument to start the program and bind it to a console.
- The abstraction can be displayed using ObjectEditor, which displays each of the main classes and allows a programmer to click on a class to invoke the open method with the class as an argument.
- A process object is wrapped in a ProcessExecer object which launches the process and stores information about the process displayed to the user

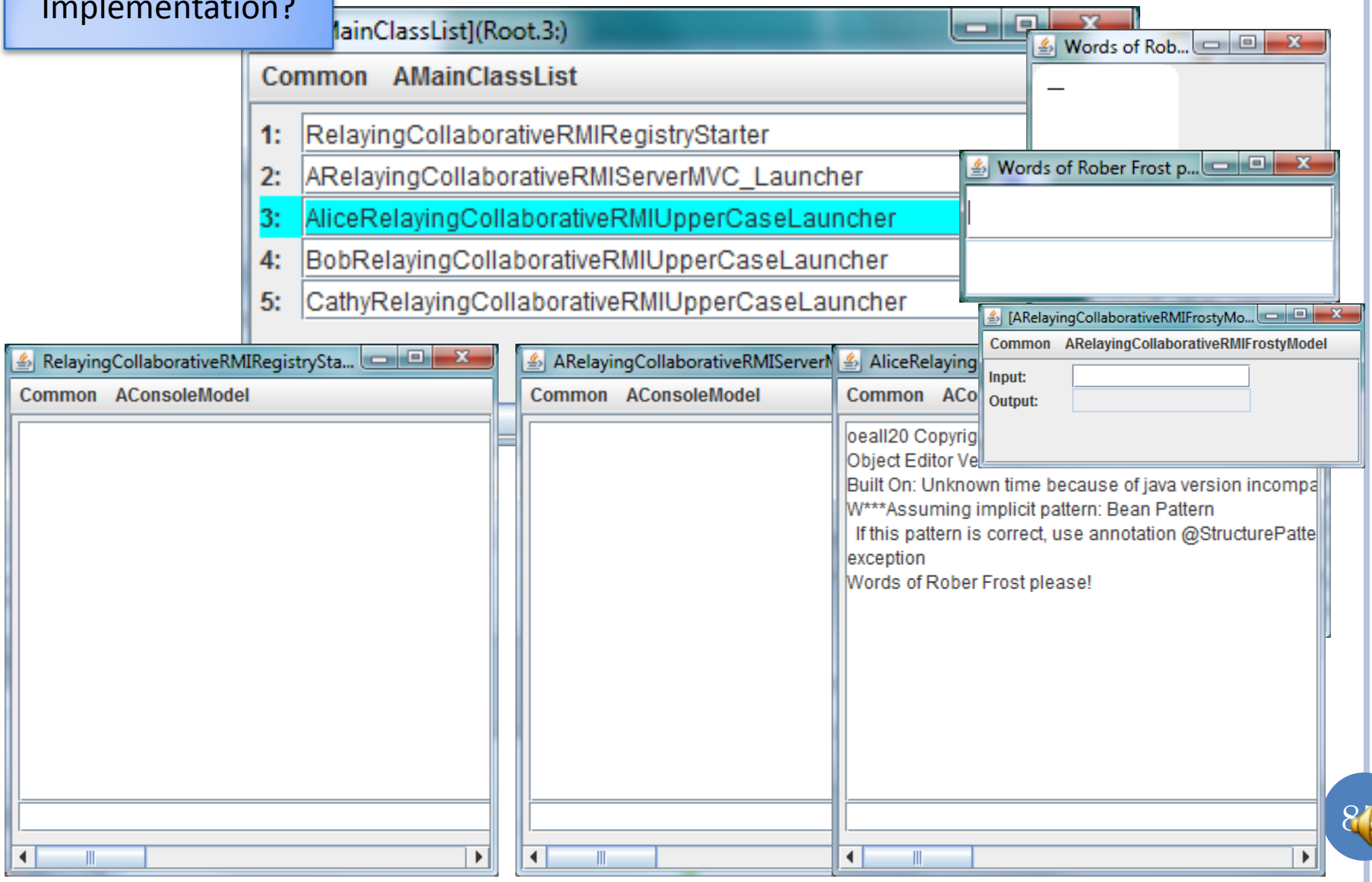
# Summary

- The console is simply a Bean with an input String property displayed in a text field and an output String property displayed in a text area.
  - It does not provide a setter to set the output directly.
  - Instead it provides an addOutput() method to append the next line of output to the console.
- Each console Bean is connected to a process whose I/O it handles
- Its setInput() method writes to the input stream of the  stream of the process.
- It creates the threads to read the processe's output and error, which call the addOutput() method

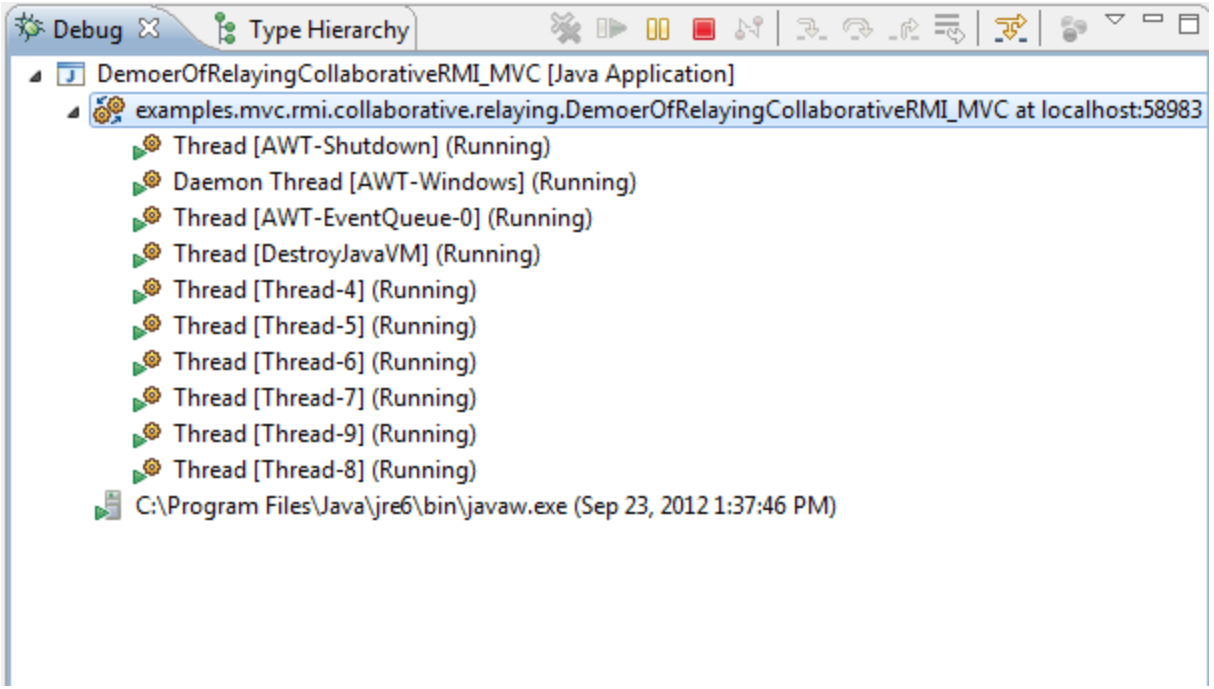
# Extra Slides

# Double clicking On Alice Launcher

Implementation?



# Double clicking On Alice Launcher



# Issue

How to use RMI for implementing various properties of a collaborative applications?

In particular one implemented using MVC?

Will use our MVC-based uppercasing application as an example

# Issue

How to use RMI for implementing various properties of a collaborative applications?

In particular one implemented using MVC?