

# Remote Method Invocation (RMI)



# Distributed Object Benefits

- Processing can be performed
  - on the most capable processor / operating system
  - where data is most easily accessible
- Can perform multi-host multithreading
  - in a network or across the internet
  - combine functionality of objects residing on many hosts into a single application
  - applications that run on different hosts can interoperate
- New applications can work with legacy applications through OO interfaces
  - such as Java classes that use native methods



# Distributed Object Issues

- Latency
  - remote calls take ~ 4 times as long as local calls
  - processor speeds are increasing faster than network speeds so this difference is unlikely to be reduced
  - design must consider which objects should be remote to get acceptable performance
- Memory Access
  - pointers are only valid within a single address space
    - not a problem for RMI for these reasons that will be understandable later
      - RMI uses object serialization to pass copies of non-remote objects and the non-remote objects reachable from them. If these non-remote objects contain references to remote objects then stubs are created for them and they can be accessed remotely.
      - RMI passes references (pointers) to remote objects. Fields of remote objects that are references to non-remote objects cannot be accessed through a remote reference because remote interfaces only expose methods, not fields.



# Distributed Object Issues (Cont'd)

- Partial Failure
  - New kinds of errors to handle
    - communication problems
      - machine crash, network failure
    - process failures
    - marshaling/unmarshaling errors
  - More difficult to support distributed transactions
    - a series of actions that must either all complete successfully or all “rollback” to return the objects involved in a transaction to their previous state
    - want to be able to lock changed objects until committed or rolled back
- Concurrency
  - multiple objects (local or remote) concurrently trying to modify and access the same remote object
    - may want to allow this in some cases
  - synchronization (locking) is needed to insure that objects being accessed are in a “valid” state
    - part of implementing transactions
  - not a problem for RMI since locking of remote objects is supported



# RMI vs. CORBA

- RMI
  - Java-only distributed object model
  - ☆ – relatively simple to use
    - especially if already familiar with Java
  - ☆ – software needed is free
    - new so lacks maturity
      - but had knowledge of CORBA
      - many RMI designers participated in the design of CORBA
- CORBA
  - ☆ – language independent distributed object model
    - but nearly all CORBA development is done in C++
  - relatively complex to use
  - software needed to use is expensive
    - but the price of OrbixWeb has been dropped to \$799
  - ☆ – spec. has been maturing for several years
    - Object Management Group (OMG)
      - started in 1989
      - over 600 member companies
    - many capabilities have been defined
      - naming service, automated launching of servers, timeouts, transactions, ...
      - vendors haven't implemented the complete set of defined object services
    - switching between ORB vendors can require major code modification



# Non-OO Approaches To Distributed Processing

- Remote Procedure Call (RPC)
  - allows invocation of remote functions
  - biggest drawback is that it's not OO
- Sockets
  - allow data transfer using of application specific protocols
    - example - cash register system
      - 1 = purchase
      - 2 = return
      - 3 = exchange
      - different data must be sent for each
  - the data transferred can indicate what should be done on the server
  - error prone due to assumptions about what types of data will be sent and received (protocol)



# RMI goals

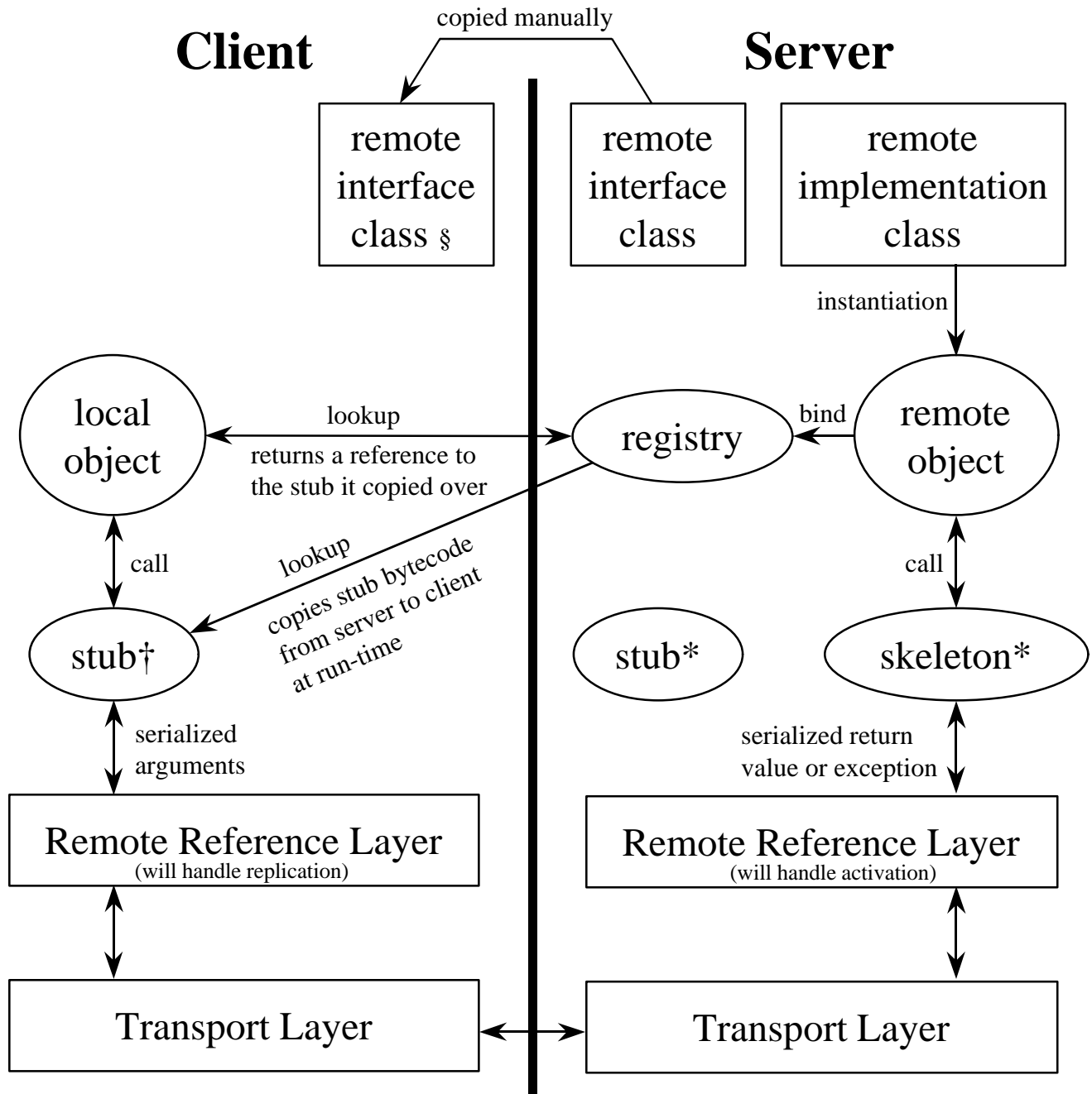
- Minimize differences between working with local and remote objects
  - but don't hide them
  - the issues presented earlier need to be considered when designing distributed applications
    - examples
      - extra exceptions must be caught for remote method calls
      - must consider which objects should be collocated
- Minimize complexity
  - make the common things easy
  - make the uncommon things doable

} common  
Java goal
- Preserve type safety
  - compile-time type checking
    - even for remote method calls
- Support distributed garbage collection
  - objects with no references in the VM may still have remote references



# RMI Operation

(descriptions of all the elements in this diagram are on pages 9-13)



§ must copy to client to compile client class OR discover through reflection

\* generated by the rmic compiler

† copied from server to client when a remote object reference is obtained





# Terminology

- **Virtual Machine (VM)**
  - a software processor with its own instruction set (Java bytecode)
  - environment in which Java methods are executed
  - one per process
  - can have more than one running on the same host
- **Client Object**
  - an object that uses the services (methods) of a remote (server) object
- **Remote/Server Object**
  - an object that provides services (methods) that can be utilized by objects in different VMs
  - implements one or more remote interfaces
  - an object can be both a client and a server object



# Terminology (Cont'd)

- Remote Interface
  - declares the methods that clients can invoke on a remote object
- Registry
  - URL-based name server used to locate *some* remote objects
    - most objects are located through method return values after the client obtains a reference to a “main” server object
  - must have one for every host/port combination where remote objects will be located via registry lookup



# Terminology (Cont'd)

- **Marshaling**

- converting remote method arguments to a stream of bytes in a stub
  - only non-remote objects are marshaled
  - for remote objects a remote reference is used
- converting the return value or exception of a remote method to a stream in a skeleton
- uses Object Serialization ←

all remote objects are Serializable because they all extend UnicastRemoteObject which extends RemoteServer which extends RemoteObject which implements Serializable

- **Unmarshaling**

- reconstructing method arguments from a stream of bytes in a skeleton
- reconstructing a method return value or exception from a stream of bytes in a stub
- uses Object Serialization ←

- **Transient Fields**

- fields of an object that are
  - not considered part of its state
  - their values may be computed at run-time
- not marshaled because they aren't serialized



# Terminology (Cont'd)

- **Stub**
  - client-side proxy for a server-side object
    - marshals arguments
    - forwards method calls to the remote reference layer
    - unmarshals return values and exceptions
  - instances are created when a reference to a remote object is passed to a client VM
    - by calling `Naming.lookup("remote object URL")`
    - by invoking a remote method that returns a remote object
  - supports the same remote interfaces as the corresponding remote object
  - code generated by the `rmic` compiler
- **Skeleton**
  - server-side object that dispatches calls to remote object methods
    - unmarshals arguments
    - calls methods in the corresponding remote object
    - marshals return values and exceptions
  - code generated by the `rmic` compiler



# Terminology (Cont'd)

- Replicated Objects
  - multiple objects in different locations that are to be kept in sync.
  - support for this is not provided in JDK 1.1
- Remote Reference Layer
  - responsible for
    - finding remote objects
    - invoking remote methods on them
    - keeping replicated objects in sync. (multicast vs. unicast)
  - can be implemented to work with server applications that
    - are always running
    - only run when a method in one of their objects is invoked
      - not available yet
- Transport Layer
  - manages communication between VMs
  - transfers serialized objects between the remote reference layers of VMs
  - informs remote reference layer of methods it should invoke

Java application that creates remote objects and adds them to the registry



# Local vs. Remote Objects

- Similarities
  - references to remote objects can be passed to and returned from methods
- Differences
  - clients can only invoke methods of remote objects that are declared in a remote interface
  - when a reference to a local object is passed to a remote method, the remote VM gets its own copy
    - methods are invoked on the local copy and changes aren't reflected in the original
    - static and transient fields are not copied since serialization doesn't include these
  - when a reference to a remote object is passed to a remote method, a stub is created in the remote VM
    - methods are invoked on the remote object, not a copy
  - all remote methods must state that they can throw `RemoteException`
  - clients must catch exceptions specific to RMI
  - cloning a stub doesn't create a new remote object



# Blocking

- Calls to remote methods block
- To achieve a non-blocking method call
  - implement the client object that will be interested in the remote method result as a remote object
  - pass that client object to the remote method
  - in the remote method
    - store the reference to the client object
    - start a thread to do the processing
    - return from call
    - when the processing completes invoke some method in the client object to notify it
  - example code demonstrates server callback to client objects



# Steps To Utilize RMI

- Create remote interfaces
  - clients will use these to invoke remote methods
- Create remote implementation classes
  - these implement the remote interfaces and provide the remote functionality
- Create server initialization application
  - creates remote objects and binds them in the registry
  - isn't needed if all remote implementation classes have a main() method that creates instances and binds them in the registry
- Create client classes
  - these can utilize remote objects
  - they locate remote objects by
    - performing a lookup in a registry
    - OR
    - invoking a method of another remote object that returns one
- Start the registries on the servers
  - allows clients to lookup remote objects
- Start the server initialization application(s)
- Start the clients

server hosts need a static IP address so clients can find them





# java.rmi Package

- interface Remote
  - all remote interfaces extend this
  - doesn't define any methods (a marker like Serializable)
- class RemoteException
  - superclass of all RMI exceptions
    - UnknownHostException
      - attempt to lookup a remote object in the registry of an unknown host
    - NotBoundException
      - attempt to lookup a name that isn't bound in that registry
    - AlreadyBoundException
      - attempt to bind to a name that was already bound to a remote object in that registry
    - StubNotFoundException
      - stub bytecode was not found on the server so it couldn't be copied to the client (and other reasons)
    - NoSuchObjectException
      - attempt to invoke a method on a remote object that is no longer available (the server application may have stopped running)
    - and many more
      - see Javasoft web page for Remote Method Invocation Specification ... Exceptions In RMI



# java.rmi Package (Cont'd)

- class Naming
  - provides methods that allow remote objects to be registered and located by URL (host, port, object name)
    - bind - associates a URL with a remote object (all objects in the same registry have the same host and port)
      - the port defaults to 1099
    - rebind - associates a previous used URL with a different remote object
    - unbind - removes the association between a URL and a remote object
    - lookup - copies a stub from the server which created the remote object associated with a given URL (host and port are used to locate the correct registry)
    - list - retrieves the URLs of all remote objects in a given registry

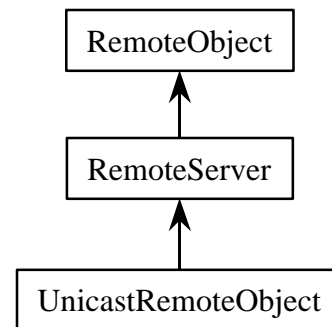


# java.rmi.server Package

- class RemoteObject
  - implements Serializable
  - superclass of all stubs and remote implementation classes
  - provides remote versions of the hashCode, equals, and toString methods in Object
    - **equals** for remote objects is overridden to be the same as == for local objects
      - for local objects
        - == tests whether two references refer to the same object
        - equals is normally overridden to test whether two objects have equal field values
      - for remote objects
        - use equals instead of == to test whether two references refer to the same object
        - write your own method to test for equality
      - reason is that remote method invocations would be required to test whether the fields of two objects are equal and the signature of equals doesn't allow for RemoteException to be thrown (all remote methods throw this)
    - **hashCode** returns the same value for all client stubs that refer to the same remote object
      - used when remote objects are used as keys in a Hashtable
    - **toString** includes the host name and port number where the remote object is registered



# java.rmi.server Package (Cont'd)

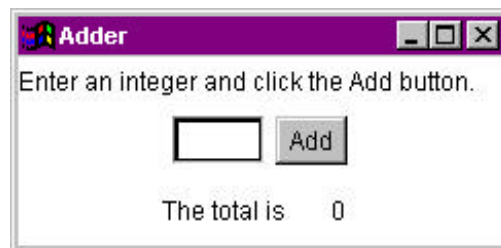


- class RemoteServer
  - extends RemoteObject
  - superclass of all remote implementation classes
  - only UnicastRemoteServer is available in JDK 1.1
  - can determine the host and port of a client that invoked a remote method in that method
    - getClientHost() and getClientPort()
- class UnicastRemoteObject
  - extends RemoteServer which extends RemoteObject
  - for non-replicated remote implementation classes
    - only one copy of each remote object
    - remote object references become invalid when the virtual machine in which they are created stops since there is only one copy of them



# Example

- A remote object maintains a total of integers
- Client objects can
  - add to the total
  - request the current total
  - request to be notified when the total changes



# Files Needed

- **Server-side**
  - Adder.java
    - remote interface for objects that maintain a total
  - AdderImpl.java
    - implements Adder
    - also an application (has a main()) that creates an object from this class and binds it in the registry
- **Client-side**
  - AdderListener.java
    - remote interface for objects that wish to be informed when the total maintained by an Adder changes
  - AdderGUI.java
    - implements AdderListener
    - client-side GUI that interacts with an AdderImpl



# Adder.java

```
import java.rmi.*;

/**
 * This interface declares the methods implemented by the
 * remote class AdderImpl.
 *
 * @author R. Mark Volkmann
 */
public interface Adder extends Remote {

    /**
     * Adds a listener to the set of listeners for this object.
     * @param listener the listener to be added
     * @exception java.rmi.RemoteException
     */
    void addListener(AdderListener l) throws RemoteException;

    /**
     * Get the current sum held by the RemoteImpl object.
     * @exception java.rmi.RemoteException
     */
    int getTotal() throws RemoteException;

    /**
     * Add a given number to the sum held by the RemoteImpl object.
     * @param x the number to add.
     * @exception java.rmi.RemoteException
     */
    void add(int x) throws RemoteException;
}
```

all remote classes  
must do this

all remote methods  
must include this  
throws clause



# AdderImpl.java

```
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
import java.util.Enumeration;
import java.util.Vector;

/**
 * This remote class maintains a total of numbers that are passed to it.
 * It can also notify listeners of changes to the total.
 *
 * @author R. Mark Volkmann
 */
public class AdderImpl extends UnicastRemoteObject implements Adder {
```

remote object implementations  
must do this

```
//-----
// Fields
//-----

private int total;
private Vector listeners = new Vector();

//-----
// Public Methods
//-----
```





# AdderImpl.java (Cont'd)

allows stub code to be loaded from a remote server  
(needed in addListener) and controls what stubs can do  
(cannot manipulate threads, exit VM, exec OS commands,  
read/write/delete local files, use sockets, access the system clipboard, etc.)

```
/**
 * Creates a remote object from this class and registers it.
 */
public static void main(String args[]) {
    System.setSecurityManager(new RMISecurityManager());
    try {
        System.out.println("creating AdderImpl");
        AdderImpl obj = new AdderImpl();
        System.out.println("binding in registry");
        Naming.rebind("//Duke/AdderServer", obj);
        System.out.println("ready");
    } catch (Exception e) {
        System.err.println(e);
    }
}
```

The application doesn't terminate when the end of main() is reached because the call to Naming.rebind() starts the distributed garbage collector in a thread and that thread is still running. The registry holds references to remote objects so they won't be garbage collected unless they are removed from the registry by calling Naming.unbind() or Naming.rebind().

```
/**
 * Constructs a new AdderImpl.
 * Even though this constructor doesn't do anything, it is required
 * so that we can specify that a RemoteException may be thrown during
 * object creation.
 * @exception java.rmi.RemoteException
 */
public AdderImpl() throws RemoteException {
}

/**
 * Adds a given number to the total.
 * @param number the number to add.
 * @exception java.rmi.RemoteException
 */
public void add(int number) throws RemoteException {
    total += number;
    notifyListeners();
}
}
```



# AdderImpl.java (Cont'd)

```
/**
 * Adds a listener to the set of listeners for this object.
 * @param listener the listener to be added
 * @exception java.rmi.RemoteException
 */
public void addListener(AdderListener listener) throws RemoteException
{
    listeners.addElement(listener);
}

/**
 * @return the current total.
 * @exception java.rmi.RemoteException
 */
public int getTotal() throws RemoteException {
    return total;
}
```



# AdderImpl.java (Cont'd)

```
/**
 * Notifies all listeners of the current total.
 * @exception java.rmi.RemoteException
 */
public void notifyListeners() throws RemoteException {
    AdderListener listener;
    Vector deadListeners = new Vector();

    Enumeration e = listeners.elements();
    while (e.hasMoreElements()) {
        listener = (AdderListener) e.nextElement();
        try {
            listener.totalChanged(total);
        } catch (UnmarshalException ue) {
            // A previous listener of this remote object
            // must have died.
            deadListeners.addElement(listener);
        }
    }

    // Remove dead listeners from the list of listeners.
    e = deadListeners.elements();
    while (e.hasMoreElements()) {
        listener = (AdderListener) e.nextElement();
        System.out.println("AdderImpl.notifyListeners: " +
            "removing a dead listener");
        listeners.removeElement(listener);
    }
}
}
```



# AdderListener.java

```
import java.rmi.*;

/**
 * An interface implemented by classes whose objects wish to be
 * notified when the total maintained by an Adder object is changed.
 *
 * @author R. Mark Volkmann
 */
public interface AdderListener extends Remote {

    /**
     * Called when the total maintained by an Adder object is changed.
     * @param total the new total
     * @exception java.rmi.RemoteException
     */
    void totalChanged(int total) throws RemoteException;
}
```



# AdderGUI.java

```
import java.awt.*;
import java.awt.event.*;
import java.net.MalformedURLException;
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;

/**
 * This remote class provides a GUI for entering numbers that are to be
 * added to a total that is maintained by another remote object, AdderImpl.
 * It receives notification of changes to the total so that even when
 * multiple instances of the GUI are running, the total shown in all of
 * them is correct.
 *
 * @author R. Mark Volkmann
 */
public class AdderGUI extends UnicastRemoteObject
implements ActionListener, AdderListener {

    //-----
    // Constants
    //-----

    private static String PROMPT =
        "Enter an integer and click the Add button.";

    //-----
    // Fields
    //-----

    private Adder adder;
    private Button addButton = new Button("Add");
    private Frame frame = new Frame("Adder");
    private Label msgLabel = new Label(PROMPT);
    private Label totalLabel = new Label();
    private TextField intField = new TextField(3);
```

needed because AdderGUI objects  
are accessed as remote objects from  
AdderImpl.notifyListeners()



# AdderGUI.java (Cont'd)

```
//-----  
// Constructors  
//-----  
  
private AdderGUI() throws RemoteException {  
    frame.add(msgLabel, BorderLayout.NORTH);  
  
    Panel panel = new Panel();  
    panel.add(intField);  
    panel.add(addButton);  
    frame.add(panel, BorderLayout.CENTER);  
  
    panel = new Panel();  
    panel.add(new Label("The total is "));  
    panel.add(totalLabel);  
    frame.add(panel, BorderLayout.SOUTH);  
  
    frame.pack();  
  
    addButton.addActionListener(this);  
    frame.addWindowListener(new WindowAdapter() {  
        public void windowClosing(WindowEvent e) {  
            System.exit(0);  
        }  
    });  
};
```



# AdderGUI.java (Cont'd)

```
try {
    // Locate the object that implements the Adder interface
    // (an AdderImpl object).
    adder = (Adder) Naming.lookup("//Duke/AdderServer");

    // Display the current total maintained by the Adder.
    totalLabel.setText(Integer.toString(adder.getTotal()));

    // Listen for changes to the total it maintains.
    adder.addListener(this);

    frame.setVisible(true);
} catch (Exception e) {
    System.err.println(e);
}

//-----
// Public Methods
//-----

/**
 * Starts the application.
 */
public static void main(String[] args) {
    System.setSecurityManager(new RMISecurityManager());

    // Since this class is remote, the constructor must be
    // declared to throw RemoteException.
    try {
        new AdderGUI();
    } catch (RemoteException e) {
        System.err.println(e);
    }
}
```

see p. 25

We don't need to add this remote object to a registry since we won't be using Naming.lookup() to find it.



# AdderGUI.java (Cont'd)

```
/**
 * Responds to the "Add" Button being clicked by sending
 * the number in the TextField to the remote Adder object.
 */
public void actionPerformed(ActionEvent e) {
    if (e.getSource() == addButton) {
        String entered = intField.getText();
        try {
            adder.add(Integer.parseInt(entered));
            msgLabel.setText(PROMPT);
        } catch (NumberFormatException nfe) {
            msgLabel.setText(entered + " is not an integer!");
        } catch (UnmarshalException ue) { ← a specific kind of RemoteException
            msgLabel.setText("AddrImpl may be dead!");
        } catch (RemoteException re) {
            System.err.println("AdderFrame.actionPerformed: " + re);
        }
    }
}

/**
 * Called when the total maintained by an Adder object is changed.
 * @param total the new total
 * @exception java.rmi.RemoteException
 */
public void totalChanged(int total) throws RemoteException {
    totalLabel.setText(Integer.toString(total));
}
}
```





# Steps to Build and Run

- To build
  - compile the source
    - `javac *.java`
  - generate stubs and skeletons
    - `rmic AdderGUI`
    - `rmic AdderImpl`
- To run
  - perform each step in a separate window to make it easy to monitor and kill each process
  - start registry (3 ways)
    - under any OS
      - `java sun.rmi.registry.RegistryImpl {port-number}`
    - under Solaris
      - `rmiregistry {port-number}`
    - under Windows
      - `start rmiregistry {port-number}`
  - start the server
    - `java AdderImpl`
  - start the clients
    - `java AdderGUI`
      - create several sessions

<ul style="list-style-type: none"><li>• port number must match what is specified in lookup URLs</li><li>• can omit to use default</li></ul>
---



# Win95/NT Additional Steps

- To run RMI, clients and servers must have fixed IP addresses
  - localhost loopback work for sockets but not RMI!
- This can be faked for PCs that are not on a network
  - the clients and servers can all run on the same machine
  - Settings...Control Panel...Network...TCP/IP...Properties...
    - DNS Configuration...
      - enter a name in the “Host:” text field
      - click the “OK” button
    - IP Address tab...
      - click the “Specify an IP address:” radio button
      - enter any IP address such as 1.2.3.4
        - note that this will make Netscape unusable
      - click the “OK” button
    - click the “OK” button
    - reboot

