JAVA REMOTE METHOD INVOCATION (RMI)

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

Client Object

Language Layer (e.g. Java Remote Method Invocation)

OS Layers (e.g. Sockets)

Network Layers (e.g. TCP/IP)

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

Server Object
REMOTE METHOD INVOCATION

Client Object

Language Layer (Java Remote Method Invocation)

Server Object
public interface Counter {
    void increment(int val);
    int getValue() throws RemoteException;
}
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();
    }
}
How do separate processes share object references?
Proxy is generated by the RMI system. 

Proxy and server objects connected through external name.
Remote method invocation has the same syntax as local method invocation.

However, caller and callee are distribution-aware.
**Checked RemoteException**

Client Object

Server Proxy

Server Object

Method call must catch RemoteException

Checked RemoteException occurs if network or server errors occur

Something the client programmer cannot control, so it is checked

Method declaration must indicate it may throw RemoteException in header

How to make programmer put the throws clause?
**Labeling Remote Methods**

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.

Programmer must label methods as remotely invokable.
**Local vs. Distributed Counter**

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

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

- Caller should handle errors
- Checked exceptions
- Distribution awareness
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();
    }
}

ACOUNTER

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

DistributedRMICounter does not extend Counter

How is it changed?
DistributedRMICounter

```java
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;
        }
    }
}
```

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

How to do client 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);
    }
}
HOW TO CREATE A LOCAL PROXY

Client Object

Server Proxy

m(p₁, ..., pᴺ)

Server Object

m(p₁, ..., pᴺ)

How to connect the two?

Proxy and server objects connected through external name and transfer of serialized proxies
**EXTERNAL NAME AND INTERNAL NAME BINDING**

<table>
<thead>
<tr>
<th>Objects shared among processes have external names.</th>
</tr>
</thead>
<tbody>
<tr>
<td>file name for files</td>
</tr>
<tr>
<td>&lt;machine, port&gt; for socket</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A mechanism is provided to bind a local reference to an external name</th>
</tr>
</thead>
<tbody>
<tr>
<td>File is opened in write mode giving file name</td>
</tr>
<tr>
<td>Server socket is bound to &lt;machine, port&gt;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A mechanism is provided to bind the external name to local reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>File is opened in read mode giving file name</td>
</tr>
<tr>
<td>Clients socket is connected to server socket using &lt;machine, port&gt;</td>
</tr>
</tbody>
</table>
Local References in Both Address Spaces

Client Object

Server Proxy

m(p₁, …, pᴺ)

Server Object

m(p₁, …, pᴺ)

How to connect the two?

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

How exactly?
**NAME SERVER**

- **Caller** registers (name, proxy) pair
- **Caller** gets object registered for name

Name server keeps (name, object) pairs
**NAME SERVER METHODS**

Remote lookup(String name)  
Registry  
rebind(String name, Remote obj)

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**

- **Remote lookup(String name)**
- **rebind(String name, Remote obj)**

**RMI Registry**

**Caller**

**Callee**

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

```java
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();
        }
    }
}
```

Usually RMIRegistry started though LocateRegistry is part of server process

Prevents termination
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)**

- Client Object
- Server Object
- Server Proxy

Method call must catch RemoteException

Method declaration must indicate it may throw RemoteException in header

Checked RemoteException occurs if network or server errors occur

Something the client programmer cannot control, so it is checked

The class of server object must implement Remote

When is proxy class created? When is proxy created?
### Generating Proxy Class: Compilation

```bash
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

<table>
<thead>
<tr>
<th>Date/Time</th>
<th>Type</th>
<th>Name</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>11/20/2011 09:12 AM</td>
<td>&lt;DIR&gt;</td>
<td>.</td>
<td></td>
</tr>
<tr>
<td>11/20/2011 09:12 AM</td>
<td>&lt;DIR&gt;</td>
<td>..</td>
<td></td>
</tr>
<tr>
<td>11/19/2011 08:17 PM</td>
<td></td>
<td>ADistributedInheritingRMICounter.class</td>
<td>933</td>
</tr>
<tr>
<td>11/20/2011 09:12 AM</td>
<td></td>
<td>ADistributedInheritingRMICounter_Stub.class</td>
<td>1,977</td>
</tr>
<tr>
<td>11/20/2011 09:13 AM</td>
<td></td>
<td>DistributedRMICounter.class</td>
<td>264</td>
</tr>
<tr>
<td>11/19/2011 07:35 PM</td>
<td></td>
<td>DistributedRMICounterClient.class</td>
<td>1,112</td>
</tr>
<tr>
<td>11/19/2011 06:17 PM</td>
<td></td>
<td>DistributedRMICounterServer.class</td>
<td>1,154</td>
</tr>
<tr>
<td>11/19/2011 08:14 PM</td>
<td></td>
<td>RMIRegistryStarter.class</td>
<td>908</td>
</tr>
</tbody>
</table>

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**

<table>
<thead>
<tr>
<th>UnicastRemote Object</th>
<th><strong>static</strong> exportObject(Remote object, int port)</th>
</tr>
</thead>
</table>

| Proxy (ADistributedInheritingCounter_Stub instance) | m(...) |

| Remote Object ADistributedInheritingCounter instance | m(...) |

**Proxy Class**

| (ADistributedInheritingCounter_Stub) | m(...) |

Creates a proxy object for the remote object at the server end that can later be sent to client

If the stub class for the proxy had not been created so far, then it is conceptually created at runtime

Stub object keeps forwarding information (host and port and id of remote object at server)
**INVALID REMOTE INTERFACE**

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

**Compilation**

```
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)
```

```
D:\dewan_backup\Java\distTeaching\bin>%javabin%\rmic
rmi.examples.AnInheritingDistributedRMICounter
error: rmi.examples.DistributedCounter is not a valid remote interface: method void increment(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.
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
Should a parameter to a remote method be passed as a proxy or a serialized copy?

- **Exported? (Remote)**
  - **Proxy**
  - **Not Serializable Exception**
    - Proxies generated by callee process through export
    - Only Remote instances can be exported

- **Serializable?**
  - **Serialized**
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)

Remote lookup(String name)

Registry

rebind(String name, Remote obj)

Caller

RMI Registry

Callee
Proxies are sent only for exported remote objects.

Exporting an object creates a proxy and creates a byte communication mechanism if such a mechanism has not been created already.
**Starting RMI Server From a Program (Review)**

```java
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();
        }
    }
}
```

Usually RMIRegistry started though LocateRegistry is part of server process.
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();
        }
    }
}
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();
        }
    }
}
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();
        }
    }
}
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);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
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);
        } catch (Exception e) {
            System.out.println(e);
        }
    }
}
Understanding Equals (Client)

```java
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.printStackTrace();
        }
    }
}
```

ADD EQUALS IN REMOTE INTERFACE?

```java
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;
}
```
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));
}
RMI Limitation

Cannot call Object methods remotely

- e.g. equals()
- e.g. toString() (ObjectEditor uses it extensively)
public class ARemoteRepository implements RemoteRepository {
    List<Remote> remotes = new ArrayList();
    public void deposit(Remote anObject) {
        remotes.add(anObject);
    }
    public List<Remote> getObjects() {
        return remotes;
    }
}
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();
        }
    }
}
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();
        }
    }
}

```
Inheriting DistributedRMICounter

```java
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**

```java
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 false;
        }
    }
}
```

- A link can be dynamically bound to reused code but requires more work.
- Correct semantics of hashcode.
- Super constructor automatically exports object.
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()
Summary: Distributed Architecture

RMIRegistry can run as part of Callee (or even caller)
**SUMMARY: RMI API**

- **Remote lookup(String name)**
- **Registry**
  - rebind(String name, Remote obj)
- **Caller**
- **Callee**
  - Registry getRegistry (String host, int port)
  - createRegistry(port)
- **LocateRegistry**
- **RMI Registry**
  - static exportObject (Remote object, int port)
- **UnicastRemote Object**
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.
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 (e.g. 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 switching away from Eclipse
LAUNCHING MULTIPLE PROCESSES: UI
DOUBLE CLICKING ON REGISTRY
DOUBLE CLICKING ON SERVER
DOUBLE CLICKING ON ALICE LAUNCHER

Implementation?

No debugger for app processes

Except for OE Code
Eclipse vs. Java Processes

How may Eclipse processes?

No debugger for app processes

All 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?
- Eclipse calls Process.destroy() and not OS kill

Regular but not eclipse process cannot
KILLING FROM AMainClassList
ALL PROCESSES KILLED

<terminated> DemoerOfRelayingCollaborativeRMI_MVC [Java Application]
<terminated> examples.mvc.rmi.collaborative.relaying.DemoerOfRelayingCollaborativeRMI_MVC at lo
<terminated, exit value: 0> C:\Program Files\Java\jre6\bin\javaw.exe (Sep 23, 2012 2:51:54 PM)
LOCAL AND REMOTE RESPONSE

Programming Interface?
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);
    }
}
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);
    }
}
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();
    }
}
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 : executed) {
        processExecer.getProcess().destroy();
    }
}
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;
}
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;
    }
}
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;
}
public Process getProcess() {
    return process;
}
public ConsoleModel consoleModel() {
    return consoleModel;
}
public void destroy() {
    process.destroy();
}
public String getTitle() {
    return title;
}
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();
    }
}
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));
}

public void addOutput(String newVal) {
    output.append(newVal + "\n");
    propertyChangeSupport.firePropertyChange(
        new PropertyChangeEvent(this, "output", null, output));
}

public StringBuilder getOutput() {
    return output;
}
public String getTitle() {
    return title;
}
public void exit() {
    process.destroy();
}
public void addPropertyChangeListener(PropertyChangeListener aListener) {
    propertyChangeSupport.addPropertyChangeListener(aListener);
}
public void initFrame(Frame aFrame) { }
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();
        }
    }
}
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 processes 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 process’s input stream.
  - Counter-intuitively a process’s input/output stream is fetched by calling the `getOutput(OutputStream)` 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 process’s output and error, which call the addOutput() method
EXTRA SLIDES
Double clicking on Alice Launcher

Implementation?
DOUBLE CLICKING ON ALICE LAUNCHER
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?