Course Name:
Advanced Java

Lecture 16
Topics to be covered

- The Roles of Client and Server
- Remote Method Invocations
- Setup for Remote Method Invocation
- Parameter Passing in Remote Methods
- Server Object Activation

What is RMI?

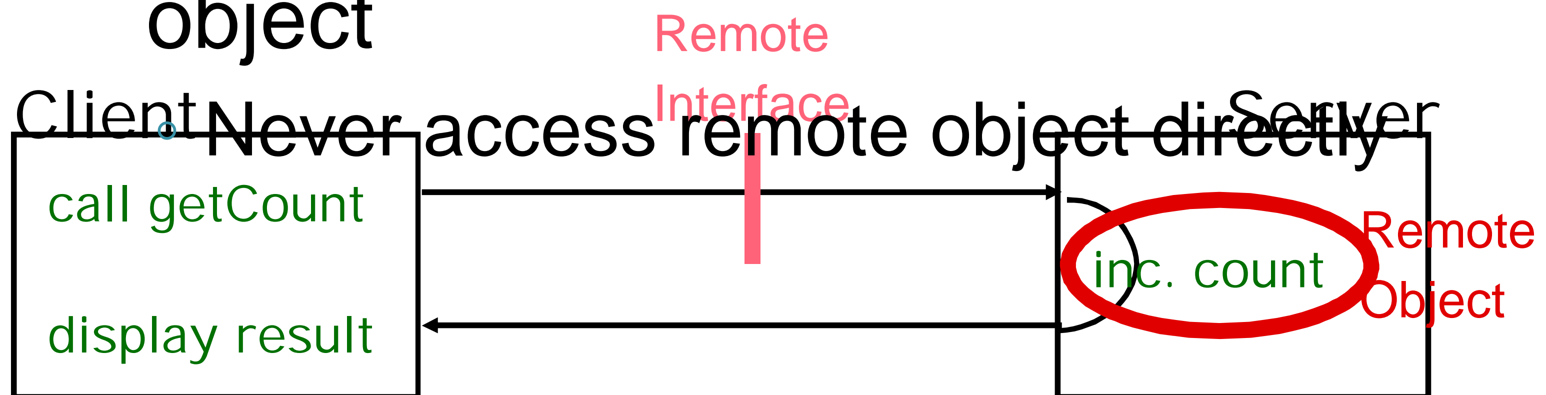
- *Java Remote Method Invocation* is a mechanism that allows calls between objects in different JVMs
- Basic concepts:
  - Remote Interface
    - defines the methods that a client can invoke on a server
  - Remote Object
    - an object whose methods can be invoked from another JVM
  - Remote Method Invocation
    - invoking a method of a remote interface on a remote object, i.e., inter-JVM call

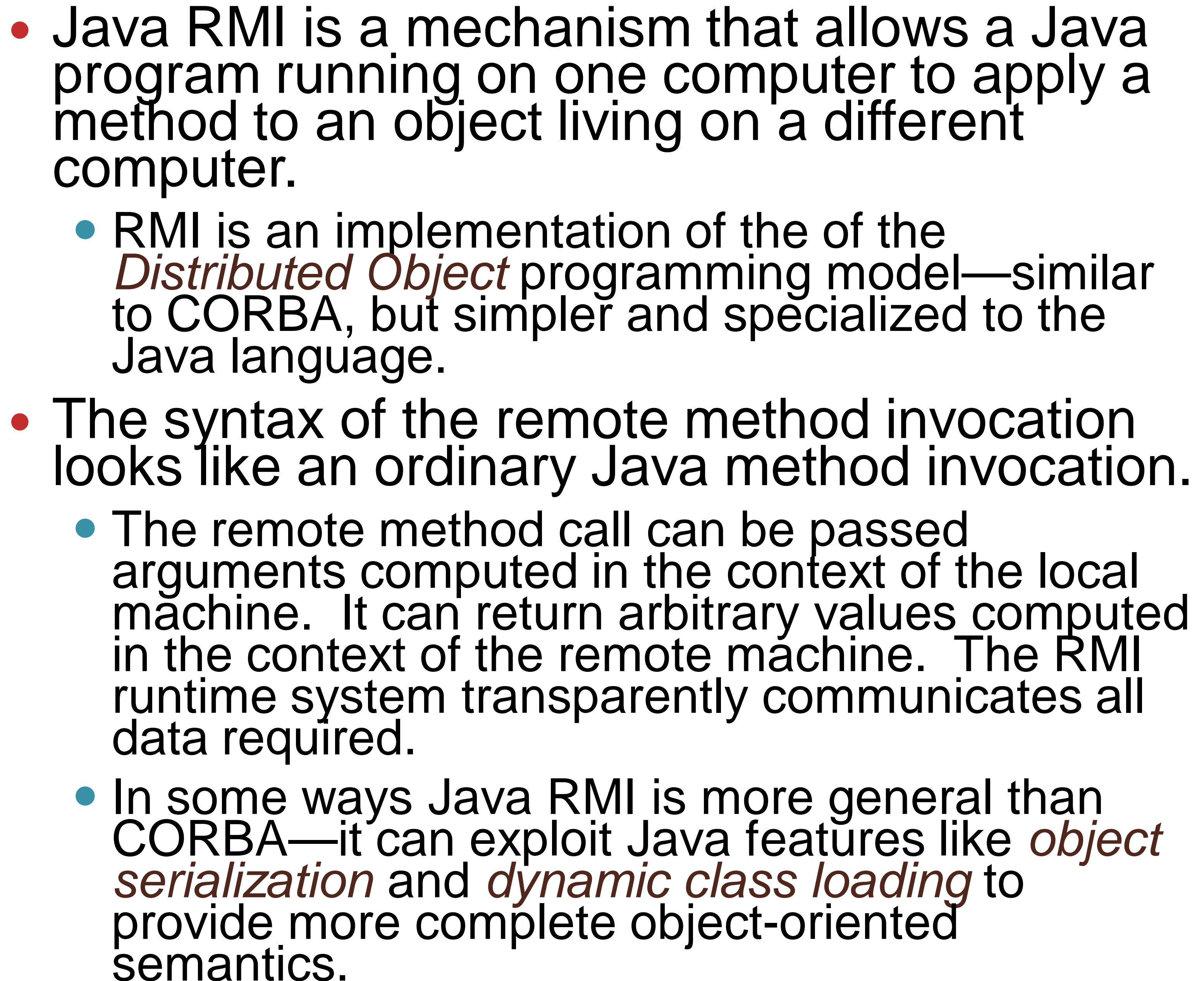
Remote Interface

- Extends java.rmi.Remote
  - java.rmi.Remote is an empty interface
  - Flags methods that can be called remotely
- Client is coded to remote interface
  - Invoking a remote method uses normal Java syntax
- All methods of a remote interface must throw java.rmi.RemoteException
  - Thrown when a remote invocation fails, e.g., a communications failure
- Used in generating stubs and skeletons

Remote Object

- Implements a remote interface
  - Can add additional methods
- Typically extends (a subclass of) java.rmi.server.RemoteObject
- Client uses a stub to refer to remote object
  - Never access remote object directly

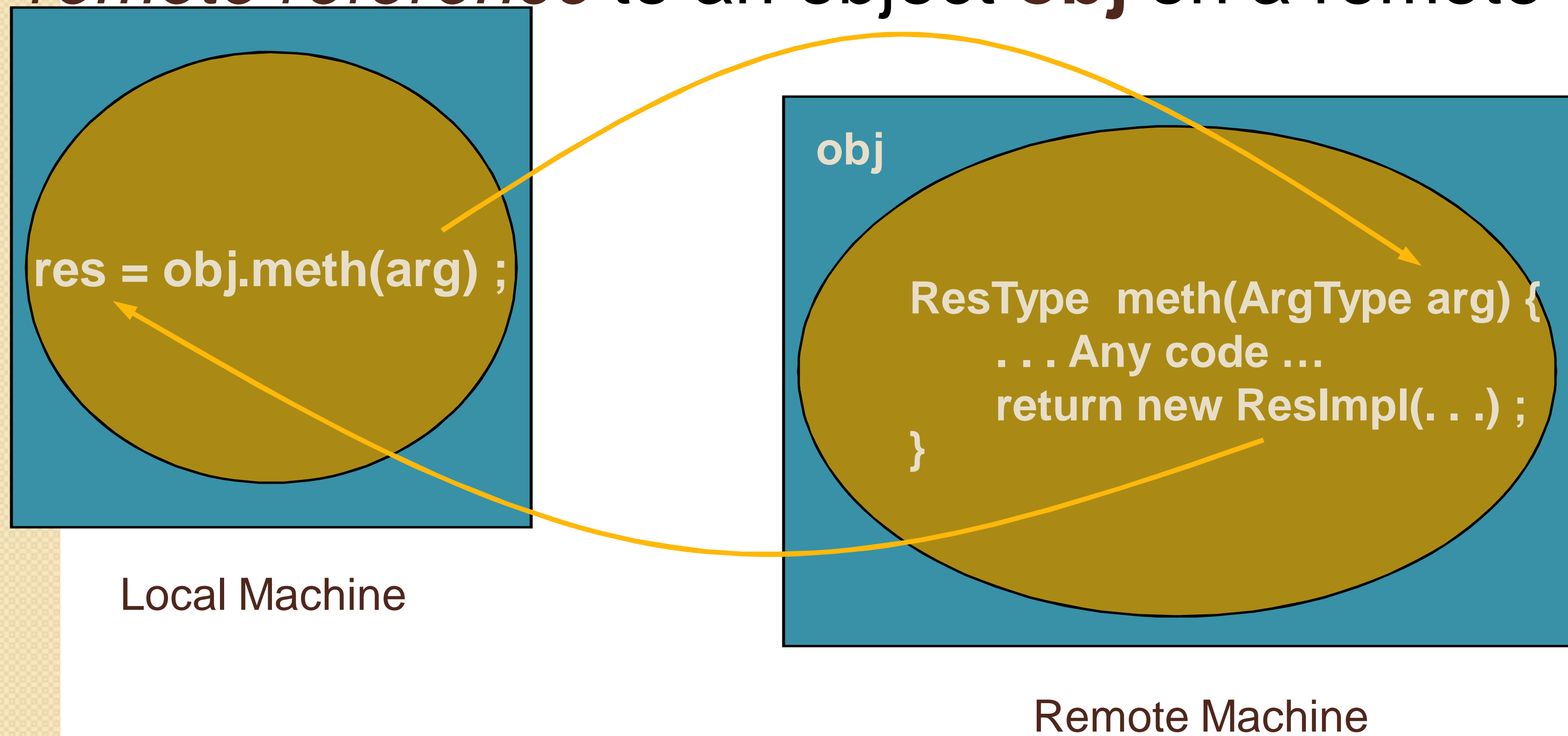




- Java RMI is a mechanism that allows a Java program running on one computer to apply a method to an object living on a different computer.
  - RMI is an implementation of the of the *Distributed Object* programming model—similar to CORBA, but simpler and specialized to the Java language.
- The syntax of the remote method invocation looks like an ordinary Java method invocation.
  - The remote method call can be passed arguments computed in the context of the local machine. It can return arbitrary values computed in the context of the remote machine. The RMI runtime system transparently communicates all data required.
  - In some ways Java RMI is more general than CORBA—it can exploit Java features like *object serialization* and *dynamic class loading* to provide more complete object-oriented semantics.

Distributed Object Picture

- Code running in the local machine holds a *remote reference* to an object **obj** on a remote



The Remote Interface

- In RMI, a common *remote interface* is the minimum amount of information that must be shared in advance between “client” and “server” machines. It defines a high-level “protocol” through which the machines will communicate.
- A remote interface is a *normal Java interface*, which must extent the marker interface **java.rmi.Remote**.
  - Corollaries: because the visible parts of a remote object are defined through a Java interface, constructors, static methods and non-constant fields are *not* remotely accessible (because Java interfaces can’t contain such things).
- All methods in a remote interface *must* be declared to throw the **java.rmi.RemoteException** exception.

A Simple Example

- A file **MessageWriter.java** contains the interface definition:

```
import java.rmi.* ;

public interface  MessageWriter
extends Remote {
        void  writeMessage(String s)
throws RemoteException ;
}
```

- This interface defines a single remote method, **writeMessage()**.

java.rmi.Remote

- The interface **java.rmi.Remote** is a *marker interface*.
- It declares no methods or fields; however, extending it tells the RMI system to treat the interface concerned as a remote interface.
  - In particular we will see that the **rmic** compiler generates extra code for classes that implement remote interfaces. This code allows their methods to be called remotely.

java.rmi.RemoteException

- Requiring all remote methods be declared to throw **RemoteException** was a philosophical choice by the designers of RMI.
- RMI makes remote invocations look *syntactically* like local invocation. In practice, though, it cannot defend from problems unique to distributed computing—unexpected failure of the network or remote machine.
- Forcing the programmer to handle remote exceptions helps to encourage thinking about how these *partial failures* should be dealt with.
- See the influential essay: "A Note on Distributed Computing" by Waldo et al, republished in The Jini Specification:

http://java.sun.com/docs/books/jini

The Remote Object

- A remote object is an instance of a class that implements a remote interface.
- Most often this class also extends the library class **java.rmi.server.UnicastRemoteObject**. This class includes a constructor that *exports* the object to the RMI system when it is created, thus making the object visible to the outside world.
- Usually you will not have to deal with this class explicitly—your remote object classes just have to extend it.
- One fairly common convention is to name the class of the remote object after the name of the remote interface it implements, but append "**Impl**" to the end.

A Remote Object Implementation Class

- The file **MessageWriterImpl.java** contains the class declaration:

```
import java.rmi.* ;
import java.rmi.server.* ;

public class MessageWriterImpl extends UnicastRemoteObject
                                                    implements MessageWriter {

    public MessageWriterImpl() throws RemoteException {
    }

    public void writeMessage(String s) throws RemoteException {
        System.out.println(s) ;
    }
}
```

Compiling the Remote Object Class

- To compile classes that implement **Remote**, you must use the **rmic** compiler. The reasons will be discussed later. For example:

```
sirah$  rmic MessageWriterImpl
```

RMI Parameter Passing

- There are two types of parameters to consider
  - Remote objects, i.e., implement java.rmi.Remote
  - Non-remote objects
- This applies both to inputs and return results

Remote Objects as Parameters

- The target receives a reference to the client stub implementing the remote interface
- Enables access to unnamed remote objects
  - Client creates a remote object and passes it as a parameter on a remote method
  - Server returns a remote object as the result of a remote method
- Enables peers and not just client-server
  - Client invokes a remote method, passing a remote object that it implements as a parameter
  - When server invokes a method on this parameter it is using a client stub, this results in a *callback* to original client

Passing Non-Remote Objects as Parameters

- Objects are passed by value
  - A copy of object is sent to the server
- Java Object Serialization used to copy parameters:
  - Non-remote-object parameters of a remote interface must be Serializable
- Use of Serialization gives different semantics than normal Java parameter passing:
  - given remote method:
    - Object identity(Object o) { return o; }
  - then:
    - o != remote.identity(o)

RMI Stubs and Skeletons

- Stubs and skeletons are mechanically generated, e.g., by *rmic* (RMI Compiler)
  - input is a class file containing a remote object, e.g., CountImpl.class
  - output is class files for stub and skeleton for the remote object
    - CountImpl_Stub and CountImpl_Skel
    - optionally can keep Java source files
  - stub class extends RemoteStub
    - stub thus has remote semantics for equals, toString and hashCode

RMI's Strengths

- Relatively easy to develop a distributed application
  - But harder than a non-distributed application
- No need to learn a separate language or object model
  - But need to learn subtle differences
- A pure Java solution
  - "Write Once, Run Anywhere"

RMI's Weaknesses

- Loss of object identity
  - If an object is passed by value, a new copy of the object is created
- Performance
  - If one is not very careful, the use of serialization can result in sending very large messages
- Potential for Deadlock if Callbacks are used
  - System *A* makes a remote call to system *B*
  - *B* makes a callback to *A*
  - The thread that will process the callback in *A* is not the thread that made the original call to *B*
  - If *A* was holding a lock when it made the initial call, deadlock may result.