RMI applications

- A typical server application creates some remote objects, makes references to them accessible, and waits for clients to invoke methods on these remote objects.

- A typical client application gets a remote reference to one or more remote objects in the server and then invokes methods on them.

RMI provides the mechanism by which the server and the client communicate and pass information back and forth.

Such an application is sometimes referred to as a distributed object application.
Design and Implement the Application Components

- *Defining the remote interfaces* => specifies the methods that can be invoked remotely by a client.

- *Implementing the remote objects* => remote objects must implement one or more remote interfaces.

- *Implementing the clients* => clients that use remote objects can be implemented at any time after the remote interfaces are defined, including after the remote objects have been deployed.
Building a generic computing engine

The compute engine server accepts tasks from clients, runs the tasks, and returns any results.

The server is comprised of an interface and a class.

The interface provides the definition for the methods that can be called from the client.

Essentially the interface defines the client's view of the remote object.

The class provides the implementation.
Designing a Remote Interface

```java
package compute;
import java.rmi.Remote;
import java.rmi.RemoteException;
// remote interface
public interface Compute extends Remote {
    // remote method
    Object executeTask(Task t) throws RemoteException;
}
```

```java
package compute;
import java.io.Serializable;
public interface Task extends Serializable {
    Object execute();
}
```
Implementing a Remote Interface

In general the implementation class of a remote interface should at least

• Declare the remote interfaces being implemented
• Define the constructor for the remote object
• Provide an implementation for each remote method in the remote interfaces

The server needs to create and to install the remote objects. This setup procedure can be encapsulated in a **main** method in the remote object implementation class itself, or it can be included in another class entirely.

The setup procedure should

• Create and install a security manager
• Create one or more instances of a remote object
• Register at least one of the remote objects with the RMI remote object if
package engine;
import java.rmi. *
import java.rmi.server. *
import compute. *

public class ComputeEngine extends UnicastRemoteObject

    implements Compute

{

    public ComputeEngine() throws RemoteException {
        super();
    }

    public Object executeTask(Task t) {
        return t.execute();
    }

    public static void main(String[] args) {
        if (System.getSecurityManager() == null) {
            System.setSecurityManager(new RMISecurityManager());
        }
        String name = "//host/Compute";
        try {
            Compute engine = new ComputeEngine();
            Naming.rebind(name, engine);
            System.out.println("ComputeEngine bound");
        } catch (Exception e) {
            System.err.println("ComputeEngine exception: "+ e.getMessage());
            e.printStackTrace();
        }
    }
}
Passing Objects in RMI

- Arguments to or return values from remote methods can be of almost any type, including local objects, remote objects, and primitive types; more precisely, any entity of any type can be passed to or from a remote method as long as the entity is an instance of a type that is a primitive data type, a remote object, or a *serializable* object (which means that it implements the interface `java.io.Serializable`).

- Many of the core classes, including those in the packages `java.lang` and `java.util`, implement the `Serializable` interface.

- The rules governing how arguments and return values are passed are as follows:
  - Remote objects are essentially passed by reference.
  - Local objects are passed by copy, using object serialization.
Compute engine = new ComputeEngine();
String name = "//host/Compute";
Naming.rebind(name, engine);

• The first parameter is a URL-formatted `java.lang.String` representing the location and the name of the remote object.

• You will need to change the value of `host` to be the name, or IP address, of your server machine.

• If the host is omitted from the URL, the host defaults to the local host.

• You don't need to specify a protocol in the URL.
Creating a Client Program

Two separate classes make up the client in our example:

✔ The first class, ComputePi, looks up and calls a Compute object.

✔ The second class, Pi, implements the Task interface and defines the work to be done by the compute engine.
package client;
import java.rmi.*;
import java.math.*;
import compute.*;
public class ComputePi {
    public static void main(String args[]) {
        if (System.getSecurityManager() == null) {
            System.setSecurityManager(new RMISecurityManager());
        }
        try {
            String name = "//" + args[0] + "/Compute";
            Compute comp = (Compute) Naming.lookup(name);
            Pi task = new Pi(Integer.parseInt(args[1]));
            BigDecimal pi = (BigDecimal) (comp.executeTask(task));
            System.out.println(pi);
        } catch (Exception e) {
            System.err.println("ComputePi exception: " + e.getMessage());
            e.printStackTrace();
        }
    }
}
package client;
import compute.*;
import java.math.*;
public class Pi implements Task {
    /** constants used in pi computation */
    private static final BigDecimal ZERO = BigDecimal.valueOf(0);
    private static final BigDecimal ONE = BigDecimal.valueOf(1);
    private static final BigDecimal FOUR = BigDecimal.valueOf(4);
    /** rounding mode to use during pi computation */
    private static final int roundingMode = BigDecimal.ROUND_HALF_EVEN;
    /** digits of precision after the decimal point */
    private int digits;
    /**
     * Construct a task to calculate pi to the specified
     * precision.
     * /
    public Pi(int digits) {
        this.digits = digits;
    }
    /**
     * Calculate pi.
     * /
    public Object execute() {
        return computePi(digits);
    }
}
/**
 * Compute the value of pi to the specified number of digits after the decimal point. The value is computed using Machin's formula:
 * \[ \pi/4 = 4\arctan(1/5) - \arctan(1/239) \]
 * and a power series expansion of \( \arctan(x) \) to sufficient precision.
 */

public static BigDecimal computePi(int digits) {
    int scale = digits + 5;
    BigDecimal arctan1_5 = arctan(5, scale);
    BigDecimal arctan1_239 = arctan(239, scale);
    BigDecimal pi = arctan1_5.multiply(FOUR).subtract(arctan1_239).multiply(FOUR);
    return pi.setScale(digits, BigDecimal.ROUND_HALF_UP);
}
/**
 * Compute the value, in radians, of the arctangent of the inverse of the supplied integer to the specified number of digits after the decimal point. The value is computed using the power series expansion for the arctangent:
 * arctan(x) = x - (x^3)/3 + (x^5)/5 - (x^7)/7 + (x^9)/9 ...
 */

public static BigDecimal arctan(int inverseX, int scale) {
    BigDecimal result, numer, term;
    BigDecimal invX = BigDecimal.valueOf(inverseX);
    BigDecimal invX2 = BigDecimal.valueOf(inverseX * inverseX);
    numer = ONE.divide(invX, scale, roundingMode);
    result = numer;
    int i = 1;
    do {
        numer = numer.divide(invX2, scale, roundingMode);
        int denom = 2 * i + 1;
        term = numer.divide(BigDecimal.valueOf(denom),
                             scale, roundingMode);
        if ((i % 2) != 0) {
            result = result.subtract(term);
        } else {
            result = result.add(term);
        }
        i++;
    } while (term.compareTo(ZERO) != 0);
    return result;
}
Build a JAR File of Interface Classes

Win32:

cd c:\home\waldo\src
javac compute\Compute.java
javac compute\Task.java
jar cvf compute.jar compute\*.class

UNIX:

cd /home/waldo/src
javac compute/Compute.java
javac compute/Task.java
jar cvf compute.jar compute/*\.class