Concurrent and Distributed Systems

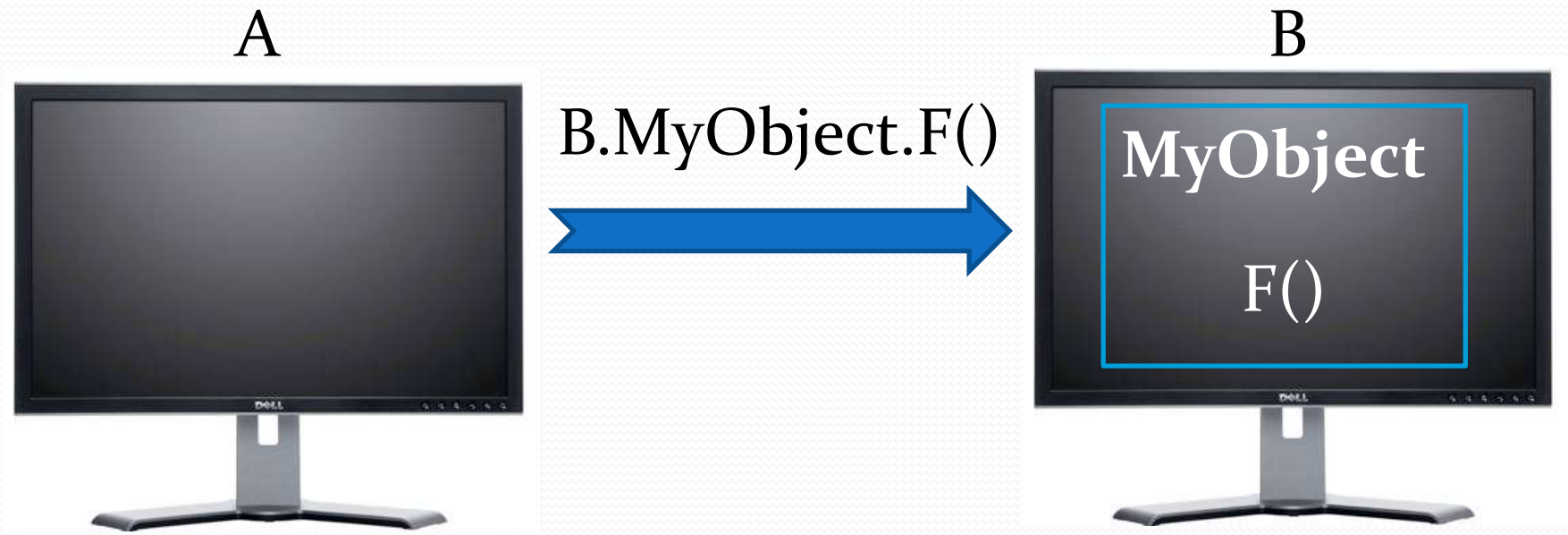
Talk Ten

Remote Calls in Java

Maxim Mozgovoy

Distributed Objects: Remote Calls

A distributed system can be built with objects, residing on different computers:



CORBA is a standard (protocol) that allows to combine objects, written in different languages and running on different computers into a single system.

CORBA vs. RMI

- CORBA: **C**ommon **O**bject **R**equest **B**roker **A**rchitecture
  - Can be used in multi-language environments.
  - Quite complicated for beginners.
- RMI: **R**emote **M**ethod **I**nvocation
  - A competing Java-only technology (can be used in Java-enabled networks).
  - Much simpler to use.

CORBA and RMI assist building client-server systems.

Instead of “clients” and “servers” there are objects with callable procedures.

Dead or Alive?

There are many CORBA-like technologies,
but today all of them are not very popular.

Generally, such systems are hard to use and confuse users by "making remote computing look like local computing" [FDC].

Also, there are more modern ways to solve the same problems.

However, in certain situations using remote objects can still be an optimal solution.

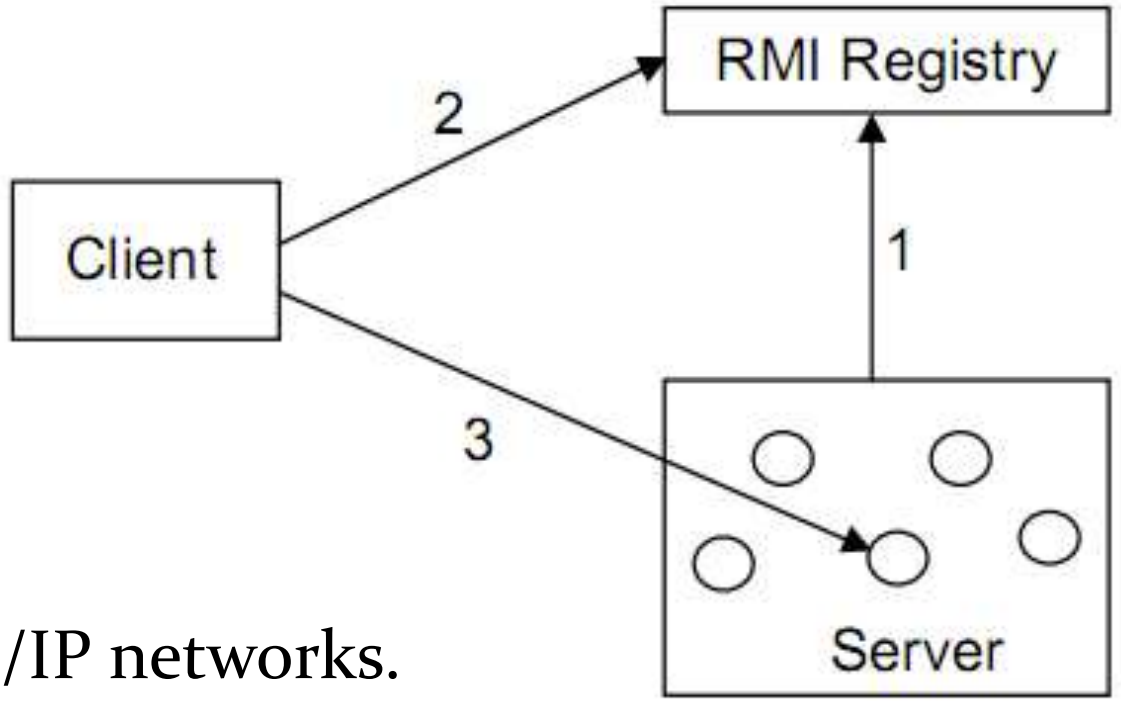
[FDC] Fallacies of Distributed Computing

1. The network is reliable.
2. Latency is zero.
3. Bandwidth is infinite.
4. The network is secure.
5. Topology doesn't change.
6. There is one administrator.
7. Transport cost is zero.
8. The network is homogeneous.

https://blogs.oracle.com/jag/resource/Fallacies.html

Architecture of RMI Programs

1. Server registers a name for its inner object in the Registry.
2. Client obtains a reference to the object from the Registry.
3. Client calls object's functions via the obtained reference.



Java RMI works in TCP/IP networks.
I.e., it assumes that the computers are addressed via IP addresses (192.168.55.1) or DNS names (mycomputer.co.jp)

“Hello, World” in RMI

Consider the following simple RMI program:

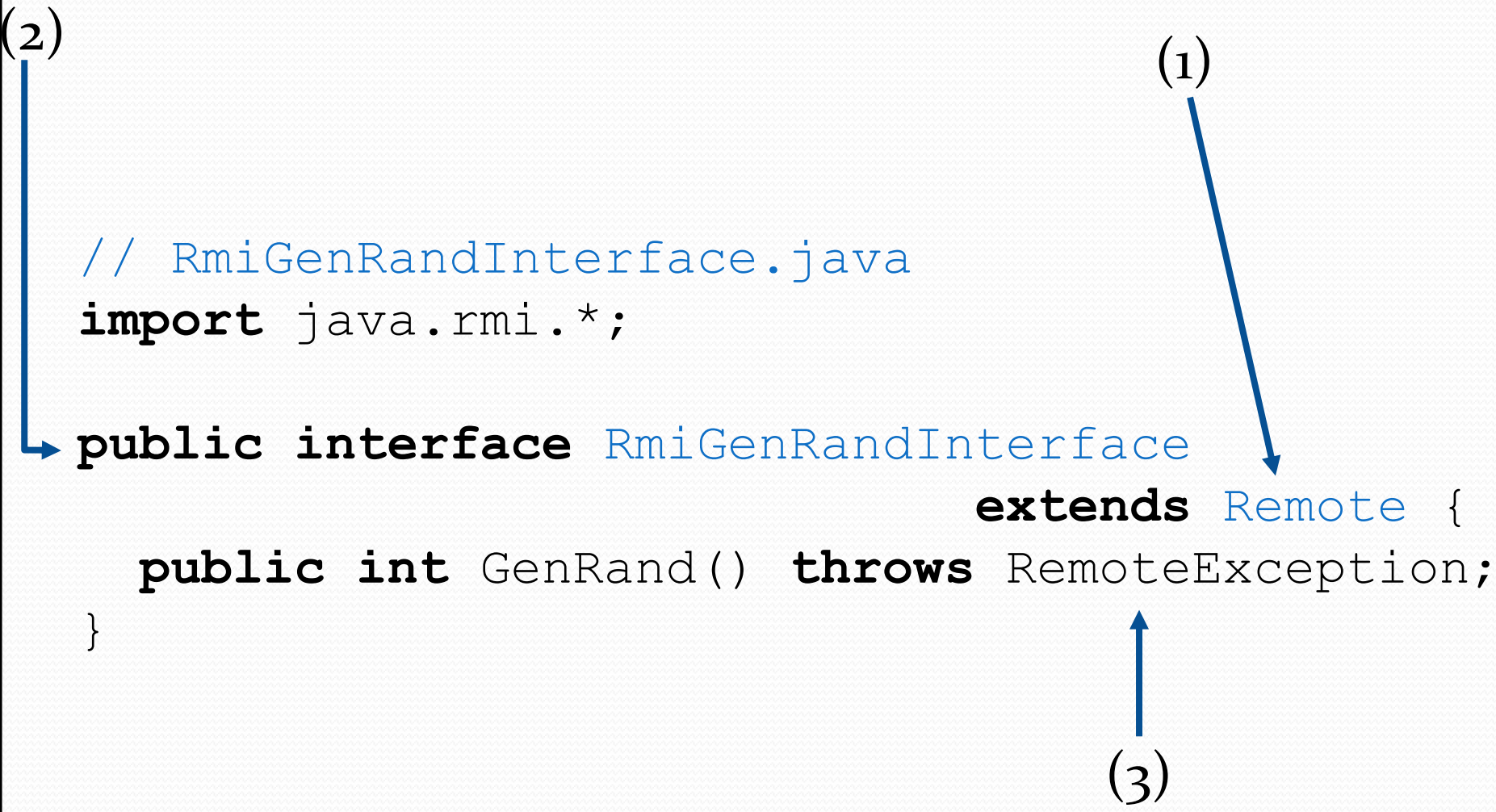
- Class RmiGenRand provides a method GenRand() that returns an integer random number.
- The server creates an object of type RmiGenRand and registers it in the RMI Registry.
- The client can obtain a reference to the remote object (using the RMI Registry) and call GenRand() to receive random numbers.

(1) Writing RmiGenRand Class

Every RMI-callable class RC should satisfy these requirements:

1. Remotely callable methods of RC should be defined in some interface RI that extends java.rmi.Remote.
2. The interface RI should be public.
3. All methods of RI should throw java.rmi.RemoteException.
4. The constructors of RC should throw java.rmi.RemoteException.
5. RC should extend java.rmi.server.UnicastRemoteObject.

(1) Writing RmiGenRand Class



(1) Writing RmiGenRand Class

(5)

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

public class RmiGenRand extends UnicastRemoteObject
                        implements RmiGenRandInterface
{
    private java.util.Random r =
                                  new java.util.Random();

    public int GenRand() throws RemoteException
                                    { return r.nextInt(); }
    public RmiGenRand() throws RemoteException {}
}
```

(4)

(2) Writing RmiGRServer Class

The server needs to register RmiGenRand object in the RMI Registry using Naming.rebind():

Naming.rebind(*RmiServiceName*, new *ClassName*(...));

Every object should have a textual name in RMI.

(2) Writing RmiGRServer Class

```
import java.rmi.*; // RmiGRServer.java

class RmiGRServer {
  public static void main (String[] args) {
    try {
      Naming.rebind("RndService",
                                  new RmiGenRand());
      System.out.println("Server is ready");
      // the server stays working!
    }
    catch (Exception e) {
        System.out.println(e);
    }
  }
}
```

(2) Writing RmiGRClient Class

```
import java.rmi.*; // RmiGRClient.java

class RmiGRClient { // print 3 random numbers
  public static void main (String[] args) {
    try {
      RmiGenRandInterface r = (RmiGenRandInterface)
           Naming.lookup("//127.0.0.1/RndService");
     System.out.println(r.GenRand());
     System.out.println(r.GenRand());
     System.out.println(r.GenRand());
    }
    catch (Exception e) {
      System.out.println(e);
    }
  }
}
```

Running Our RMI Program

1. Run RMI Registry:

```
rmiregistry &
```

Says nothing (the Unix way ☺)

1. Run the server:

```
java RmiRGServer &
```

Says "Server is ready"

1. Run the client:

```
java RmiRGClient
```

Prints random numbers, e.g.:
-1045375538
602450640
-225386443

In MS Windows rmiregistry should be run in a separate shell, and with SET CLASSPATH=""

Separating the Files

How this program looks like if the server and the client are run on different computers?

- Server: RmiGenRand.class, RmiGenRandInterface.class, RmiGRServer.class

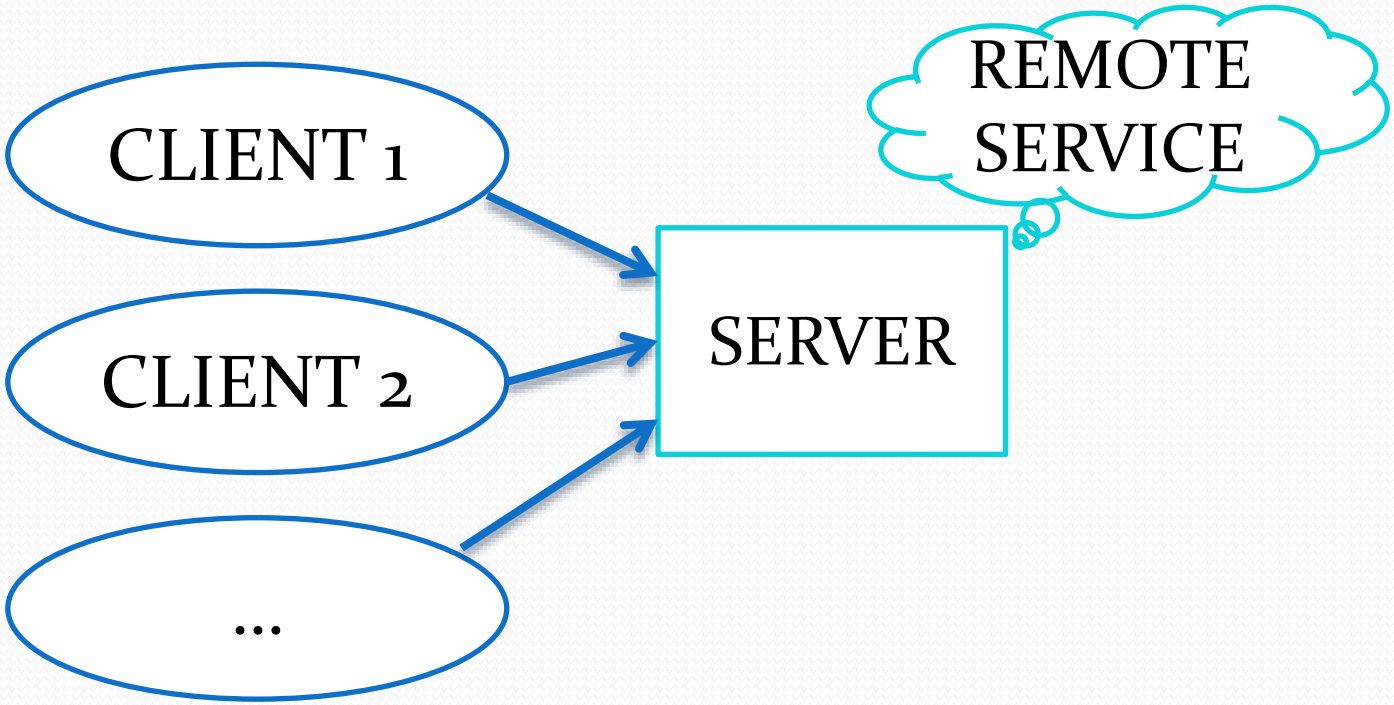
- Client: RmiGenRandInterface.class, RmiGRClient.class

That's why we need a separate *interface* file:
we need to supply it to a client.

There is no need to distribute RmiGenRand class. It's good a) for security reasons; b) because it makes updates easier.

Handling Multiple Requests (1)

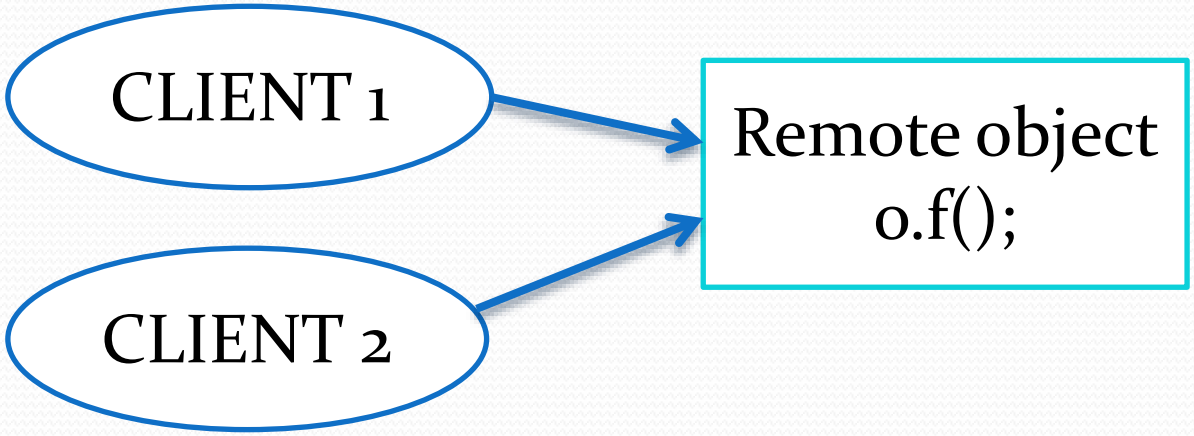
RMI handles multiple requests automatically!



(This means that we can run several clients at the same time)

Handling Multiple Requests (2)

But RMI is not thread safe:



These two requests can be concurrent (be executed in parallel).

(But there is only one instance of the remote object, of course)

So we should care about synchronization ("thread safety").

Old New Money Example

CLIENT 1

```
MoneyI m =
(MoneyI)Naming.lookup("…");

for(int i=0; i<200000; ++i)
    m.AddMoney();

System.out.println("Acc: "
            + m.GetMoney());
```

```
int money = 0;

void AddMoney() {
    int a = money;
    a++;
    money = a;
}

int GetMoney() {
    return money;
}
```

CLIENT 2
(THE SAME)

Typical printout:
(Client 1) Acc: 380958
(Client 2) Acc: 399992

Old New Money Example: How to Fix

```
public class Money extends UnicastRemoteObject
                   implements MoneyInterface  {
  private int money = 0;

  public Money() throws RemoteException {}

  synchronized public void AddMoney()
                        throws RemoteException {
    int a = money;
    a++;
    money = a;
  }

  public int GetMoney() throws RemoteException {
        return money;
  }
}
```

Typical printout:
(Client 1) Acc: 382217
(Client 2) Acc: 400000

Additional Notes

- RMI Registry must be run on the same host as the server process. In other words, RMI registry works as a “façade” for this specific server’s services only.

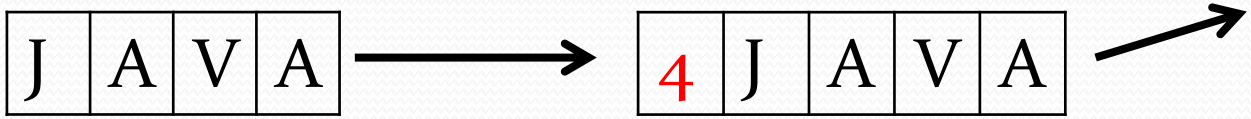
- Custom objects passed to and from remote methods must implement Serializable interface.

What is Serialization?

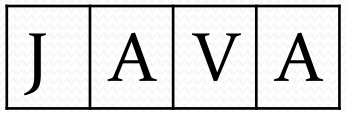
Serialization is the process of converting an in-memory object into a stream of bytes for storage on a disk or transmission over a network.

Deserialization is a reverse process of reconstructing an in-memory object from a stream of bytes.

Ex: storing a string.



L = ReadByte()
Allocate an L-byte array
Read L elements



Implementing serialization/deserialization can be tricky for non-trivial types (objects referencing other objects, complex data structures, etc.)

Serialization in Java & RMI (1)

In Java, most built-in classes can be serialized automatically.

Suppose instead of one random number in RmiGenRand
I want to generate several numbers.

Changes are simple:

```
// RmiGenRandInterface.java
import java.util.ArrayList;
import java.rmi.*;

public interface RmiGenRandInterface
                                    extends Remote {
  public ArrayList GenRand(int N)
                        throws RemoteException;
}
```

Serialization in Java & RMI (2)

```
import java.util.ArrayList; // RmiGenRand.java
import java.rmi.*;
import java.rmi.server.*;

public class RmiGenRand extends UnicastRemoteObject
                        implements RmiGenRandInterface
{
    ... // same as before
    public ArrayList GenRand(int N)
                                  throws RemoteException {
        ArrayList ret = new ArrayList();
        for(int i = 0; i < N; ++i)
            ret.add(r.nextInt());
        return ret;
    }
}
```

Serialization in Java & RMI (3)

```
import java.rmi.*; // RmiGRClient.java
import java.util.ArrayList;

class RmiGRClient { // print 3 random numbers
  public static void main (String[] args) {
    try {
      RmiGenRandInterface r = (RmiGenRandInterface)
           Naming.lookup("//127.0.0.1/RndService");
      ArrayList values = rnd.GenRand(5);
      for(var v : values)
          System.out.println(v);
    }
    catch (Exception e) {
      System.out.println(e);
    }
  }
}
```

Serializing Custom Classes (1)

```
public class RandomPack {
    public RandomPack(int iv, float fv) {
        IntValue = iv; FloatValue = fv;
    }
    public int IntValue;
    public float FloatValue;
}
...
public interface RmiGenRandInterface extends Remote {
  public RandomPack GenRand() throws RemoteException;
}

// Will generate an exception when running:
// java.io.NotSerializableException: RandomPack
```

Serializing Custom Classes (2)

```
import java.io.Serializable;

public class RandomPack implements Serializable {
    public RandomPack(int iv, float fv) {
        IntValue = iv; FloatValue = fv;
    }
    public int IntValue;
    public float FloatValue;
}
```

Serialization: From Magic to Data

"[Serialization] was a horrible mistake in 1997"

--Mark Reinhold - Chief Architect, Java Platform Group

Serialization *as designed in Java* can be used to perform custom code execution, which leads to security holes. Same issues were found in other similar tools (like Python pickle module).

Read also: tinyurl.com/owasp-ud

Modern trend:

1) Do not serialize *objects*, serialize *data* that can be used to reconstruct objects with receiver's existing functionality.
2) Use standardized formats like JSON or XML to store data.

However, the issues of thread safety, race condition, etc.
are still applicable and must be addressed.

Spiritual Successors: REST Services

In modern development, RESTful (Representational State Transfer) web services are used more often than remote calls.

Such services are based on several principles, including:

- Separation into “client” and “server” sides.
- Statelessness: no client context is stored on the server side; all state information is specified inside the request object*.
- Service identification with a URI address.
- Communication with a service using conventional HTTP commands, such as GET and POST.
- Use of standardized formats (like JSON) for data transfer*.

*Notable differences from RMI.

Services are hard to develop in plain Java,
the use of additional frameworks like Spring is necessary.

Using REST

Calling *simple* REST services from standard Java is easy (though typically we need 3rd party libraries in most cases)

Usually:

- A REST service is accessible via a conventional URL.
- A URL is a combo of a "base URL" and a "function" we call.
- Function name is followed by a "?" character.
- Function arguments are passed as "arg=value" pairs.
- Arguments are separated with the "&" symbol.
- Argument values must be escaped (transformed) so that certain special symbols are passed correctly.
- Accessing a REST service might require authentication.

Example: Joke API (1)

Documentation: https://sv443.net/jokeapi/v2/

Can be used to retrieve *[mostly silly]* jokes using a public API.

For example, to retrieve programming-related jokes consisting of two parts in the text format, we need to call:

Base URL: https://v2.jokeapi.dev/joke
Function: Programming
Arguments: format: "txt"
type: "twopart"

Thus, the resulting URL to be accessed is:

https://v2.jokeapi.dev/joke/Programming?format=txt&type=twopart

This URL can be simply opened in a browser to retrieve a joke.

Example: Joke API (2)

In Java, we can use the following code:

```
import java.net.URL; import java.io.*;
import java.lang.*;
String call_url(String base_url, String function,
                String[] args) throws Exception {
    String argslist = String.join("&", args);
    URL url = new URL(base_url + "/" + function
                      + "?" + argslist);
    InputStreamReader is =
          new InputStreamReader(url.openStream());
    int c; String r = "";
    while((c = is.read()) != -1)
        r += (char)c;
    return r;
}
```

Example: Joke API (3)

Let's call it:

```
System.out.println(call_url(
      "https://v2.jokeapi.dev/joke",
      "Programming",
      new String[]{"format=txt", "type=twopart"}
));
```

Example result:

```
What do you call a group of 8 Hobbits?

A Hobbyte.
```

Example: Datamuse API (1)

Documentation: https://www.datamuse.com/api

"A word-finding query engine for developers".

For example, to retrieve words that are spelled similar to "supaman", we need to access:

Base URL: https://api.datamuse.com

Function: words

Arguments: sp: "supaman"

Thus, the resulting URL will be:
https://www.datamuse.com/api/words/?sp=supaman

The result will be retrieved in JSON format (check www.json.org)

Example: Datamuse API (2)

```
System.out.println(call_url(
        "https://api.datamuse.com",
        "words", new String[]{"sl=supaman"}));
```

Result:

[{"word":"superman","score":90,"numSyllables":3},{"word":"superme
n","score":90,"numSyllables":3},{"word":"super
man","score":85,"numSyllables":3},{"word":"saponin","score":82,"nu
mSyllables":3}, ...]

This text can be later transformed into an array of simple objects like

```
class Entry {
    public String word;
    public int score;
    public int numSyllables;
}
```

Conclusions

- RMI makes remote object calls almost as simple as local calls.
- We simply call obj.f(), while obj may belong to another computer in the TCP/IP network.
- The client only needs access to the object's interface declaration and to the RMI Registry.
- The arguments of obj.f() and its return value are passed via network (which is slow). So frequent send / receive operations of large objects should be avoided.
- The process of converting objects into a stream of data is called serialization. Nowadays we typically use formats like JSON.
- There are no true "servers" and "clients" in RMI. All modules can create remote objects and pass them to each other.
- RMI is not thread-safe. Remote methods may be executed in parallel, so we should care about synchronization.
- A more modern alternative to RMI is REST API.