Distributed Systems

Summer Term 2020

Roland Wismüller
Universität Siegen
roland.wismueller@uni-siegen.de
Tel.: 0271/740-4050, Büro: H-B 8404

Stand: May 25, 2020

Distributed Systems

Summer Term 2020

3 Distributed Programming with Java RMI

Content

- Introduction
- *Hello World* with RMI
- RMI in detail
  - classes and interfaces, stubs, name service, parameter passing, factories, callbacks, ...
- Deployment: loading remote classes
  - Java remote class loader and security manager

Literature

- WWW documentation and tutorials from Oracle
  - `http://docs.oracle.com/javase/6/docs/api/index.html?java/rmi/package-summary.html`
  - `http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136424.html`
- Hammerschall: Ch.. 5.2
- Farley, Crawford, Flanagan: Ch. 3
- Horstmann, Cornell: Ch. 5
- Orfali, Harkey: Ch. 13
- Peter Ziesche: Nebenläufige & verteilte Programmierung, W3L-Verlag, 2005. Ch. 8

3.1 Introduction

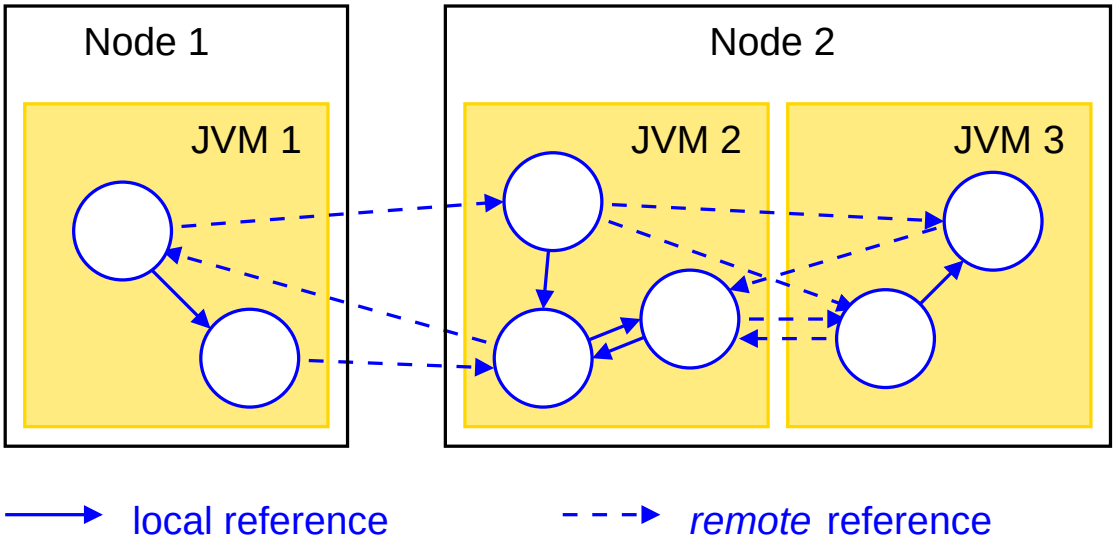
- Java RMI is an integral part of Java
  - allows use of remote objects
- Elements of Java RMI:
  - remote object implementations
  - client interfaces (stubs) to remote objects
  - server skeletons for remote object implementations
  - name service to locate objects in the network
  - service for automatically creating (activating) objects
  - communication protocol
- Java interfaces for the first five elements
  - in the package `java.rmi` and its subpackages

3.1 Introduction ...

- Java RMI requires that all objects (i.e., client and server) are programmed in Java.
  - in contrast to, e.g., CORBA
- Advantage: seamless integration into the language
  - use of remote objects is (almost!) identical to local objects
  - including distributed garbage collection
- Integration of objects in other programming languages:
  - “wrapping” in Java code via Java Native Interface (JNI)
  - use of RMI/IIOP: interoperability with CORBA
    - direct communication between RMI and CORBA objects

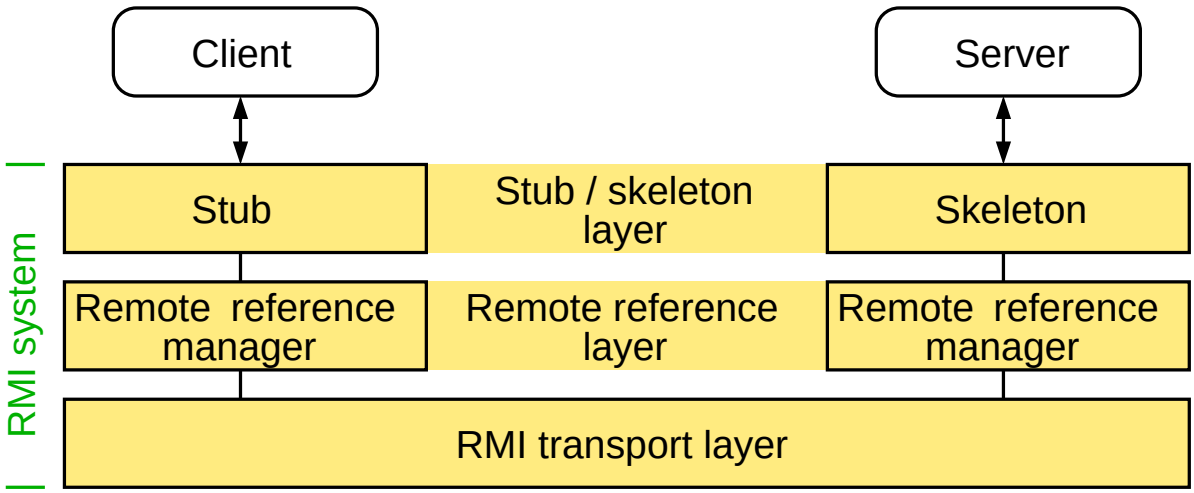
(Animated slide)

Distributed Objects



- Remote references can be used just like local references
- Objects can occur in client and server roles

3.1.1 RMI Architecture



Stub/Skeleton Layer

- Stub: local proxy object for the remote object
- Skeleton: receives calls and forwards them to the correct object
- Stub and skeleton classes are automatically generated from an interface definition (Java interface)
- As of JDK 1.2: skeleton class is generic
  - skeleton uses reflection mechanism of Java to call methods of server object
  - reflection allows you to query the method definitions of a class and to generically call methods at runtime
- As of JDK 1.5: stub classes are created at runtime
  - with the Java class `Proxy`

Remote Reference Layer

- Defines call semantics of RMI
  - in JDK 1.1: unicast only, point-to-point
    - call is routed to exactly one existing object
  - as of JDK 1.2 also activatable objects
    - object will be (re-)activated first, if necessary
      - new object, state is restored from hard disk
  - also possible: multicast semantics
    - proxy sends request to a set of objects and returns the first response
- Also: connection management, distributed garbage collection

Transport Layer

- Connections between JVMs
  - basis: TCP/IP streams
- Proprietary protocol: Java Remote Method Protocol (JRMP)
  - allows tunneling the connection via HTTP (due to firewalls)
  - allows you to define your own socket factory, e.g. to use Transport Layer Security (TLS or SSL)
- As of JKD 1.3 also RMI-IIOP
  - uses IIOP (Internet Inter-ORB Protocol) from CORBA
  - thus: direct interoperability with CORBA objects

3.1.2 RMI Services

- **Name service: RMI Registry**
  - registers remote references to RMI objects under freely selectable unique names
  - a client can then get the corresponding reference for a name
    - technical: registry sends serialized proxy object (client stub) to the client.
    - the location of the required `class` files may also be transferred (see **3.4.1**)
  - RMI can also be used with other naming services, e.g. via JNDI (Java Naming and Directory Interface)

3.1.2 RMI Services ...

- **Object Activation Service**
  - usually: remote reference to RMI object is only valid as long as the object exists
    - if the server or the server JVM crashes: object references become invalid
      - references change on restart!
  - RMI Activation Service introduced with JDK 1.2
  - starts server objects on request of a client
    - server object must register an activation method with the RMI Activation Daemon

3.1.2 RMI Services ...

- **Distributed Garbage Collection**
  - automatic garbage collection of Java also works for remote objects
  - server-side JVM manages a list of remote references to objects
  - references are “leased” for a certain time
  - reference counter of the object is decremented, if
    - client deletes the reference (e.g., end of the lifetime of the reference variable), or
    - client does not renew the lease in time
      - reason: remote reference layer cannot explicitly “log off” an object, if the client crashes
      - default setting: 10 min.

Structure:

Client JVM

Server JVM

Interface

```
interface Hello {
  String sayHello();
}
```

Client class

```
class HelloClient {
  ...
  Hello h;
  ...
  s = h.sayHello();
...
```

Server class

```
class HelloServer
      implements Hello {
  String sayHello() {
    return "Hello World";
  }
...
```


Development Process:

1. Design the interface for the server object
2. Implement the server class
3. Develop the server application to include the server object
4. Develop the client application with calls to the server object
5. Compile and start the system

Designing the Interface for the Server Object

- Specified as normal Java interface
- Must extend `java.rmi.Remote`
  - no inheritance of operations, only marking as remote interface
- Each method must declare to raise the exception `java.rmi.RemoteException` (or a base class of it)
  - base class for all errors that may occur
    - in the client, during transmission, in the server
- No restrictions compared to local interfaces
  - but: semantic differences (parameter passing!)

Hello-World Interface

```
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface Hello extends Remote {
    String sayHello() throws RemoteException;
}
```

Marker interface, contains no methods, marks interface as RMI interface

RemoteException indicates error in the remote object or during communication

Implementing the Server Class

- A class that is to be usable remotely must:
  - implement a given remote interface
  - usually extend `java.rmi.server.UnicastRemoteObject`
    - defines call semantics: point-to-point
  - have a constructor that declares to throw a `RemoteException`
    - creation of object must be done in a `try-catch` block
- Methods usually do not need to specify `throws RemoteException`
  - because they don't throw the exception themselves

Hello-World Server (1)

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

public class HelloServer extends UnicastRemoteObject
                         implements Hello {
  public HelloServer() throws RemoteException {
     super();
  }
  public String sayHello() {
    return "Hello World!";
  }
```

Remote method

Development of the Server Application to Include the Server Object

- Tasks:
  - creating a server object
  - registering the object with the name service
    - under a specified public name
- Typically not a new class, but `main` method of the server class

Hello-World Server (2)

```
  public static void main(String args[]) {
    try {
      HelloServer obj = new HelloServer();
      Naming.rebind("rmi://localhost/Hello-Server", obj);
    }
    catch (Exception e) {
      System.out.println("Error: " + e.getMessage());
      e.printStackTrace();
    }
  }
}
```

Create the server object

Register the server object under the name "Hello−Server" with the name server (RMI registry, local host, port 1099)

Development of the Client Application with Calls to the Server Object

- Client must first use the name service to get a reference to the server object from the name service
  - type cast to the correct type required
- Then: any method can be called
  - no syntactical differences to local calls
- Note: client can get remote references in other ways as well
  - e.g. as return value of a remote method

Hello-World Client

```
import java.rmi.*;

public class HelloClient {
  public static void main(String args[]) {
    try {
      Hello obj =
        (Hello)Naming.lookup("rmi://bspc02/Hello-Server");
      String message = obj.sayHello();
      System.out.println(message);
    }
    catch (Exception e) {
      ...
    }
  }
}
```

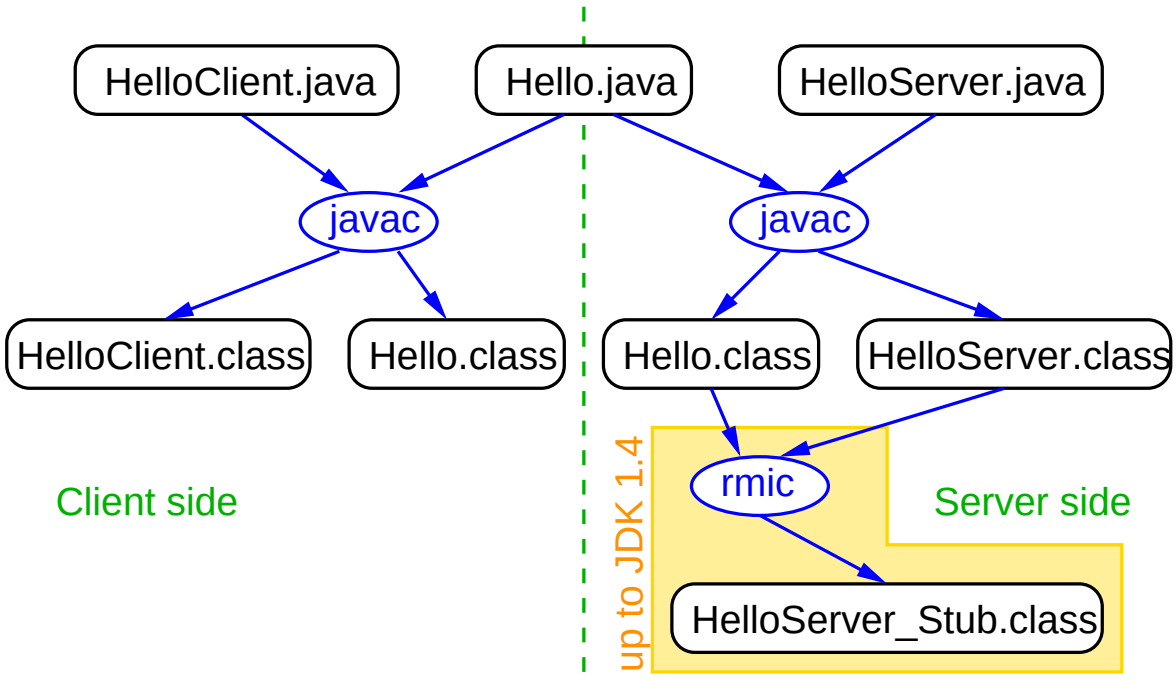
Get object reference from name server

Call the method on the remote object

Compiling and Starting the System

- Compiling the Java sources
  - source files: `Hello.java`, `HelloServer.java`, `HelloClient.java`
  - invocation: `javac *.java`
  - creates `Hello.class`, `HelloServer.class`, `HelloClient.class`
- Creating the client stub (proxy object)
  - for clients up to JDK 1.4:
    - invocation: `rmic -v1.2 HelloServer`
    - creates `HelloServer_Stub.class`
  - as of JDK 1.5: client creates proxy class at runtime
    - using `java.lang.reflect.Proxy`

Compiling and Starting the System ...



Compiling and Starting the System ...

- Starting the naming service
  - invocation: `rmiregistry` *[port]*
  - for security reasons, objects can only be registered on the local host
    - i.e. RMI registry must run on server computer
  - standard port: 1099
- Starting the server
  - invocation: `java HelloServer`
- Starting the client
  - invocation: `java HelloClient`

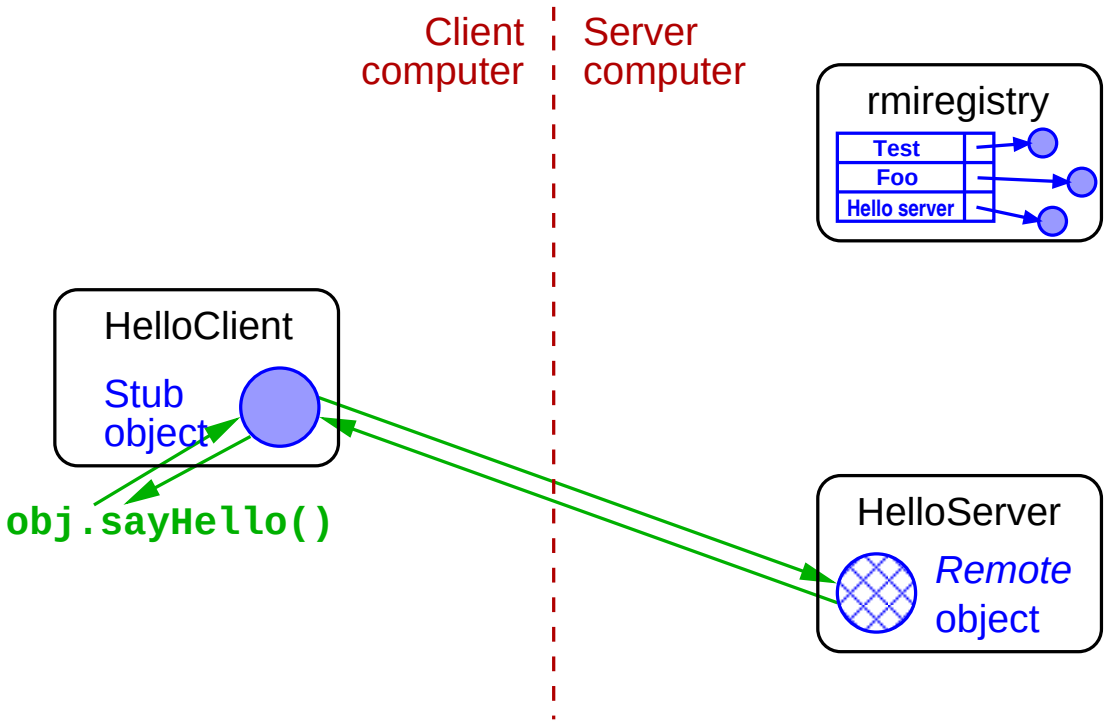
Notes for slide 121:

The example assumes that the class `Hello.class` (and, if applicable, also `HelloServerStub.class`) are found using the local classpath:

- when starting `rmiregistry`
- when starting `HelloServer`
- when compiling and starting `HelloClient`

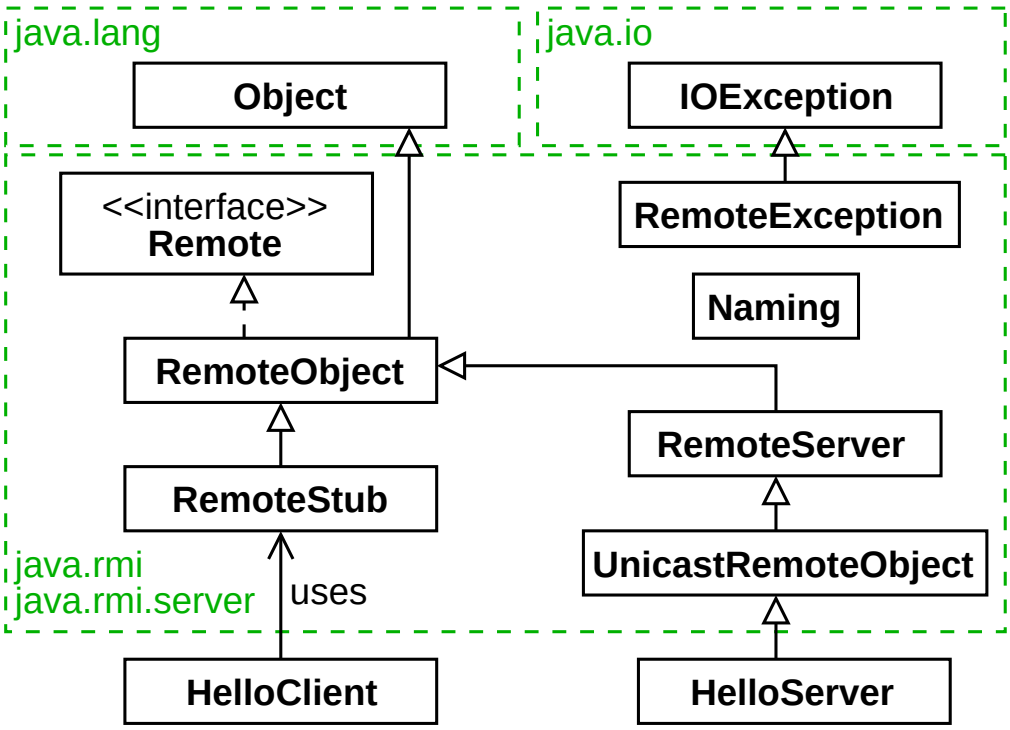
(Animated slide)

Execution of the Example



3.3 RMI in Detail

3.3.1 Classes and Interfaces



3.3.1 Classes and Interfaces ...

Interface `Remote`

- Every remote object must implement this interface
- Does not provide methods, serves only as a marker

Class `RemoteException`

- Superclass for all exceptions that can be triggered by the RMI system, for example, with
  - communication errors (server not reachable, ...)
  - (un-)marshalling errors
  - protocol errors
- Each remote method must specify `RemoteException` (or a base class of it) in the `throws` clause

3.3.1 Classes and Interfaces ...

Class `RemoteObject`

- Base class for all remote objects
- Redefines the methods `equals`, `hashCode`, and `toString`
- `toStub()` returns a reference to the stub object
- `getRef()` returns remote reference (= Java class)
  - used by the stub to call methods

Class `RemoteServer`

- Base class for all server implementations
  - `UnicastRemoteObject`, `Activatable`
- Method `getClientHost()`: host address of the client of the current RMI call
- `setLog()` and `getLog()`: logging of RMI calls

3.3.1 Classes and Interfaces ...

Class `UnicastRemoteObject`

- Implements remote object with the following properties:
  - references to the object are only valid as long as server process (JVM) is still running
  - client call is routed to exactly one object (via TCP connection), no replication
- Constructor allows definition of port and socket factories
  - so that e.g. connections via TLS/SSL can be realized
- Static method `exportObject()` makes object available via RMI
- Static method `unexportObject()` cancels availability

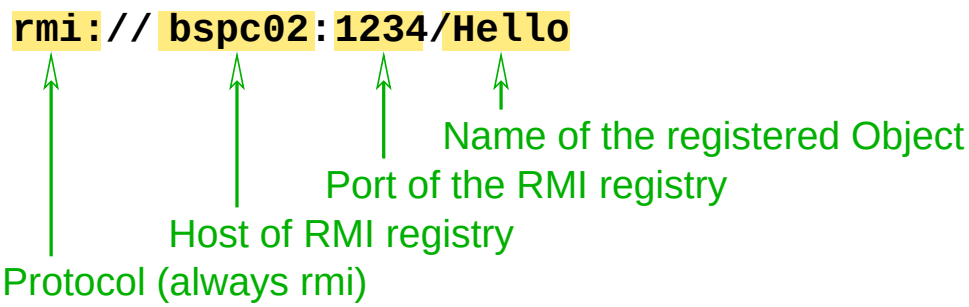
Class `RemoteStub`

- Base class for all client stubs

3.3.1 Classes and Interfaces ...

Class `Naming`

- Allows easy access to RMI registry
- Important methods:
  - `bind()` / `rebind()`: registers object under given name
  - `lookup()`: get object reference to a name
- Names are given in URL format
  - also define the host and port of the RMI registry.
  - structure of the URL:



Notes for slide 127:

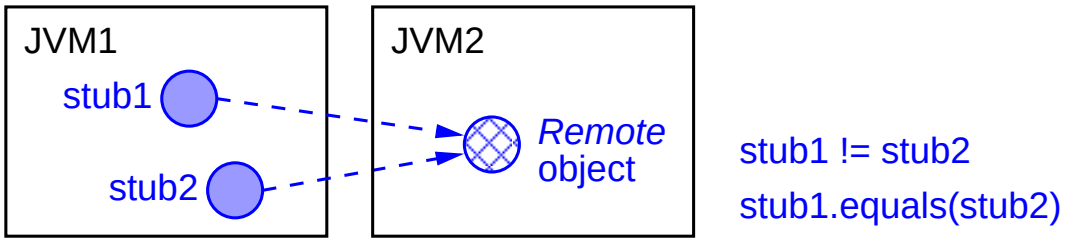
The method `rebind()` overwrites an existing entry with the same name, while `bind()` throws an exception in this case.

Another more flexible way to access the RMI registry is to use the `LocateRegistry` class and the `Registry` interface in the `java.rmi.registry` package.

3.3 RMI in Detail ...

3.3.2 Special Features of Remote Classes

- Comparison of remote objects
  - Comparison with `==` refers only to the stub objects
    - Result is `false`, even if both stubs refer to the same remote object
  - comparison with `equals()` returns true if both stubs refer to the same remote object



3.3.2 Special Features of Remote Classes ...

- Method `hashCode()`
  - used by container classes `HashMap`, `HashSet` and others
  - Hash code is calculated only from the object identifier of the remote object
    - same remote object ⇒ same hash code
    - but the content of the object is ignored
  - consistent with behavior of `equals()`
- Cloning objects
  - cloning of the remote object is not possible by calling `clone()` on the stub
  - cloning of stubs neither necessary nor meaningful

3.3 RMI in Detail ...

3.3.3 Parameter Passing

- Parameters passed to remote methods
  - either via *call-by-value*
  - or via *call-by-reference*
- The mechanism used depends on the type of the parameter
- Final decision may only be made at runtime!
- The return of the result follows the same rules as for parameter passing

Parameter Passing for Local Methods

- Java supports two kinds of types:
  - **value types**: simple data types
    - `boolean`, `byte`, `char`, `short`, `int`, `long`, `float`, `double`
    - are passed to local methods *by value*
    - that is, the method receives a copy of the value
  - **reference types**: classes (incl. `String` and arrays)
    - are passed to local methods *by reference*
    - that is, the method works on the original object and can also change object if required

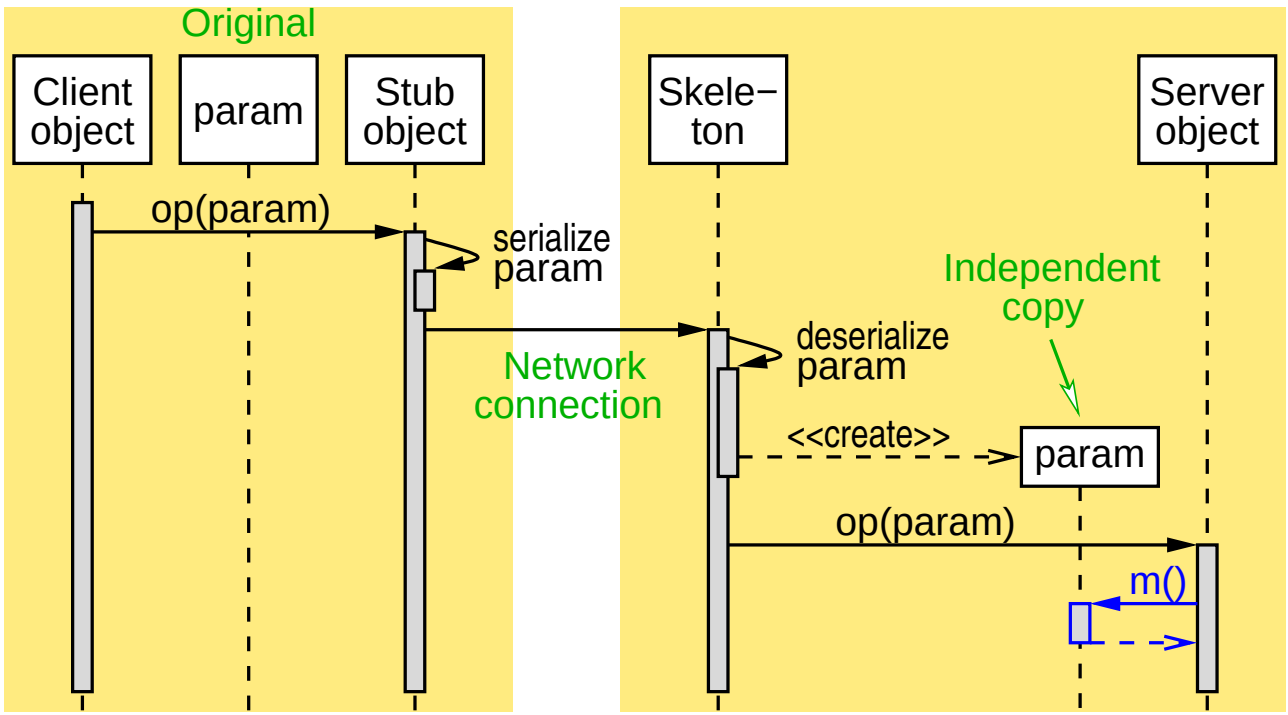
Parameter Passing for Remote Methods

- Value types: are always passed *by value*
- Reference types: dependent on the concrete object
  - object can be serialized: *call-by-value*
  - object belongs to a class that implements the `Remote` interface: *call-by-reference*
  - neither: error (`java.rmi.MarshalException`)
  - both: ??! (this case is to be avoided!)
  - decision is made only at runtime

Call-by-Value (Serializable Objects)

- Class must implement interface `java.io.Serializable`
- Serializable objects can be transferred over a network
  - only the data is transferred, the code (`class` file) must be available at the receiver!
- Default serialization of Java:
  - all attributes of the object are serialized and transferred
  - recursive procedure!
  - prerequisite: all attributes and all base classes can be serialized
- Application specific serialization is possible:
  - implement the methods `writeObject` and `readObject`

Passing a Serializable Object



Call-by-Reference (Remote Objects)

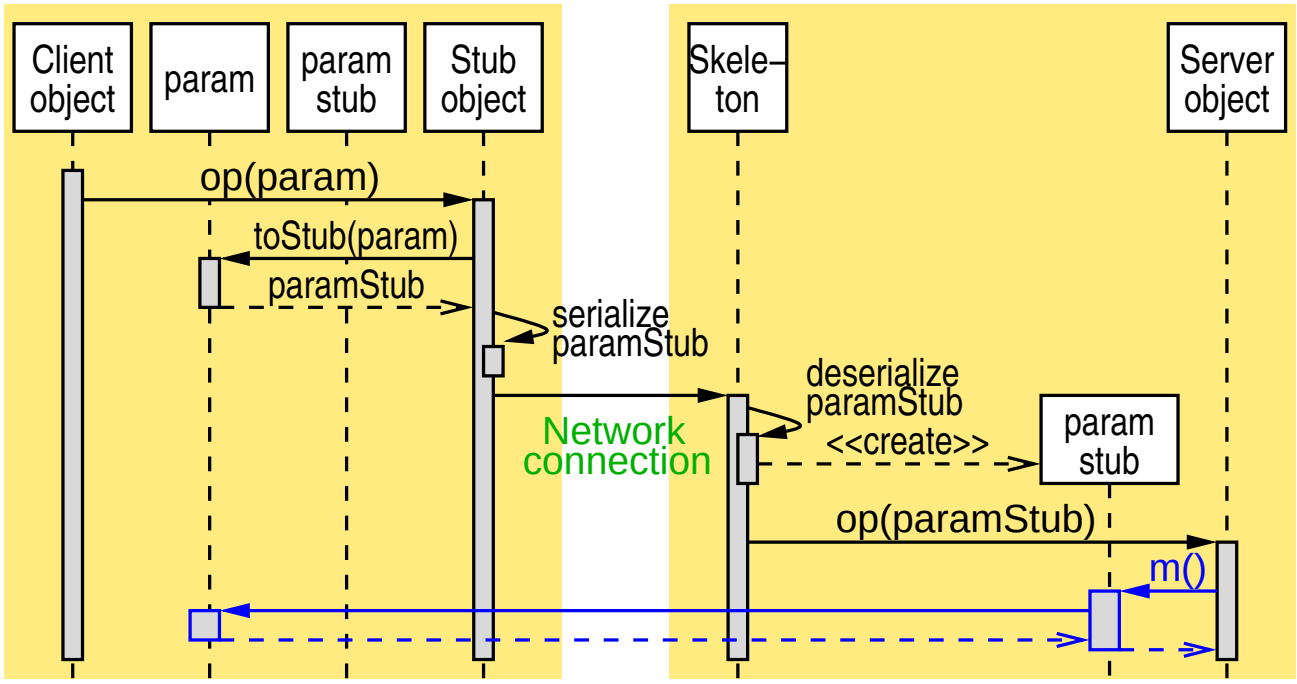
- Class of the parameter object must implement an interface that extends `Remote`
  - parameter type must be this interface
  - class is typically derived from `UnicastRemoteObject`
- A serialized stub object is transferred
  - from JDK 1.5: stub class is created dynamically
  - (up to JDK 1.4 stub class must be generated by `rmic` and be available at the server)
- If the server calls methods on the parameter object:
  - calls are routed to the original object using RMI

Notes for slide 135:

More precisely, starting with JDK 1.5, the receiver dynamically creates the stub class if the **sender** cannot load a stub class created with `rmic`.

The reason for this behavior is that during serialization, information about how the class was loaded at the sender is also transferred. If the sender has loaded the class locally from a file, the receiver also receives this information and then also tries to load the class locally (although it could just as well create it dynamically). See also slide 152.

Passing a Remote Object



Examples

- See WWW:
  - *Hello-World* with *call-by-value* parameter
  - *Hello-World* with *call-by-reference* parameter

Arrays and Container Objects

- Arrays and container objects (from the Java Collection Framework, `java.util`) can be serialized
  - i.e., they will be reinstantiated at the receiver
- To the elements of the array / container the same rules apply as to simple parameters
  - for mixed content: elements are passed *by value* or *by reference* depending on their actual class

3.3.4 Remote Object References as Results

- Frequently: via RMI registry, the client receives a reference to a remote object, which provides references to other objects
  - the remote object may also create these objects on demand (this is called *factory object* or *factory class*)
- Example: server for bank accounts
  - registration of all account objects with RMI registry not useful
  - instead: registration of a manager object that returns the reference to the account object for a given account number
    - if necessary, it can create a new object (from a database)
- Note: RMI does not allow remote object creation
  - client cannot create objects on a remote host

3.3.5 Client Callbacks

- Frequently: server wants to make calls in the client
  - e.g. progress bar, queries, ...
- For this: client object must be an RMI object
  - pass `this` reference to the server method
- In some cases, you cannot inherit from `UnicastRemoteObject`, e.g. for applets
  - then: export the object using `UnicastRemoteObject.exportObject(obj,0);`
  - attention: when calling `exportObject(obj)`, no dynamic stub is created, even with JDK 1.5 and later
- Example code: see WWW (*Hello-World* with callback)

Notes for slide 140:

The second parameter of `exportObject()` is the port on which the server object listens. A 0 means to choose any free port, just like the variant with only one parameter does.

3.3.6 RMI and Threads

- RMI does not specify how many threads are provided on the server side for method calls
  - only one thread, one thread per call, ..
- This means that several server methods can be active at the same time
  - requires correct synchronization (`synchronized`)!
- Client-side locking of a remote object using a `synchronized` block is not possible
  - only local stub is locked
  - a lock must be implemented using methods of the remote object if necessary

3.4 Deployment

- **Deployment**: distribution, transfer and installation of the components of a distributed application
  - specifically for RMI: which `class` file has to go where?

- **Server**, **RMI registry** and **client** need the `class` files for:
  - the remote interface of the server
  - all classes or interfaces that are used in the server interface (recursively)
  - (up to JDK 1.4 also the stub classes for all used remote interfaces)

- **Client** and **server** additionally need the class files for:
  - their own implementation
  - all classes of serializable objects that they receive
    - as a parameter or a result of method calls
  - (up to JDK 1.4 also the stub classes for all remote objects they receive)
- Problems with static installation of `class` files for serialized objects (and stubs):
  - dependency between client and server
    - method parameters, result objects
  - change of classes requires new installation
    - nullifies an advantage of distributed applications

3.4.1 Remote Class Loading in Java RMI

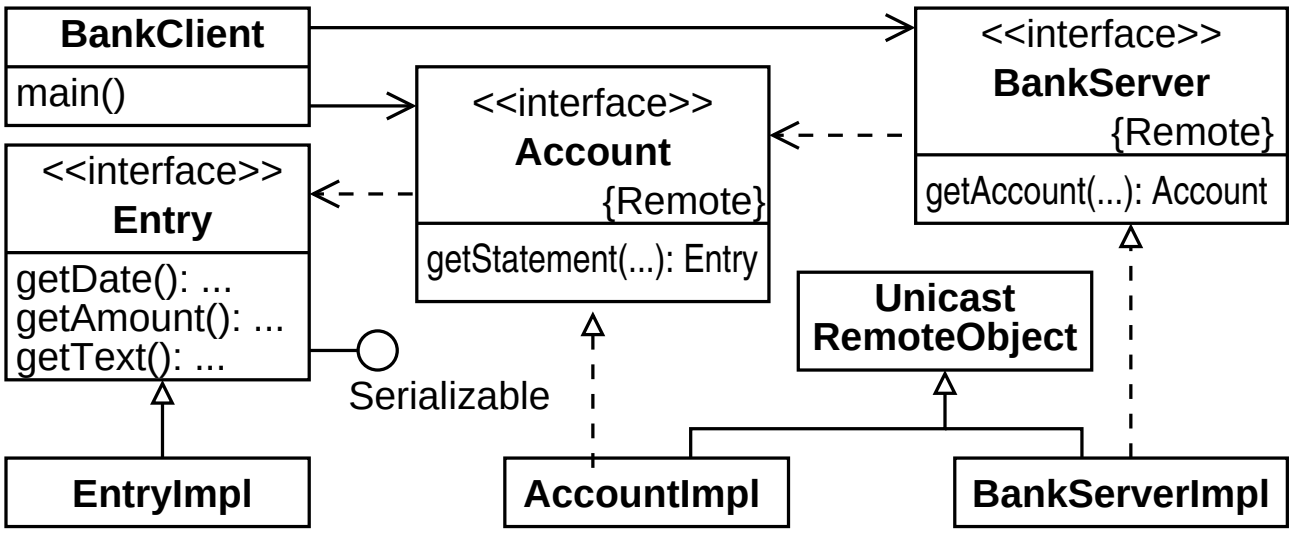
Class loader

- Class loaders are used for loading classes (and interfaces) at runtime
  - more exactly: for loading `class` files
- Each class is loaded only once
- Class loaders are Java objects themselves
  - base class: `java.lang.ClassLoader`
- RMI uses its own class loader
  - `java.rmi.RMIClassLoader`

Remote Loading of Classes

- `RMIClassLoader` allows to load classes also from remote computers
  - via HTTP (web server) or FTP
  - URL is defined via `codebase` property when the JVM is started
- Allows central storage of the necessary files
  - “automatic” deployment
- Restrictions:
  - all classes named in the client code must be available locally
  - client must define its own security manager

Example



- The `class` files for `BankServer`, `Account` and `Entry` must be available locally at the client (`BankClient`)
- `EntryImpl` can be remotely loaded by client

Local and Remote Loadable Classes

- Local loading (via `CLASSPATH`) must be possible (for client and server) for:
  - all classes (and interfaces) named in the client/server code, all classes mentioned by name in those classes, ..
  - i.e. everything that is needed to compile the code
- Remotely loadable:
  - stub classes of remote objects
  - subclasses accessed only via polymorphism
    - i.e. the code only uses a superclass or interface
- The RMI registry can load all required classes remotely

Example: *Hello-World* with Callback and Result Object

- Interfaces (see WWW):

```
public interface Hello extends Remote
{
  HelloObj getHello(AskUser ask) throws RemoteException;
}

public interface AskUser extends Remote
{
  boolean ask(String question) throws RemoteException;
}

public interface HelloObj
{
  void sayIt();
}
```

Example: How are the Classes Loaded?

- Interfaces `Hello.class`, `AskUser.class`, `HelloObj.class`
  - must be available locally at the client
  - can be loaded remotely by the RMI registry
- Implementation `HelloObjImpl.class` of `HelloObj`
  - can be loaded remotely by the client
  - is not required by RMI registry
- Stub classes for the two remote interfaces
  - are usually generated dynamically (not loaded) as of JDK 1.5
  - but can also be loaded remotely

Example: Necessary Changes in the Client

- Using the RMI `security manager`:

```
public static void main(String args[]) {
  System.setSecurityManager(new RMISecurityManager());
```

- Definition of security policy: *policy file*:

```
grant {
  permission java.net.SocketPermission "myserver:1024-",
             "connect,accept";
  permission java.net.SocketPermission "www.bsvs.de:80",
             "connect";
};
```

  - grants local classes (client!) the following permissions:
    - connection to/from myserver on non-privileged ports:
      - RMI registry (1099), server and callback object (dyn.)
    - connection to the web server (port 80)

Example: *Deployment*

- All classes to be loaded remotely are packed into one archive
- The archive is made available via a web server
- Start the server with a `codebase`, e.g:
  - `java -Djava.rmi.server.codebase="http://www.bsvs.de/jars/HelloServer.jar" HelloServer`
  - the `codebase` property specifies the URL to the JVM under which the classes are to be loaded
  - server passes `codebase` to RMI registry when registering the server object
  - RMI registry passes codebase to client
- Start of the client with specification of the policy file, e.g:
  - `java -Djava.security.policy=policy HelloClient`

Notes for slide 151:

The RMI registry must not find the classes to be loaded remotely locally (via the `CLASSPAT` otherwise it does not pass the `codebase` to the client.

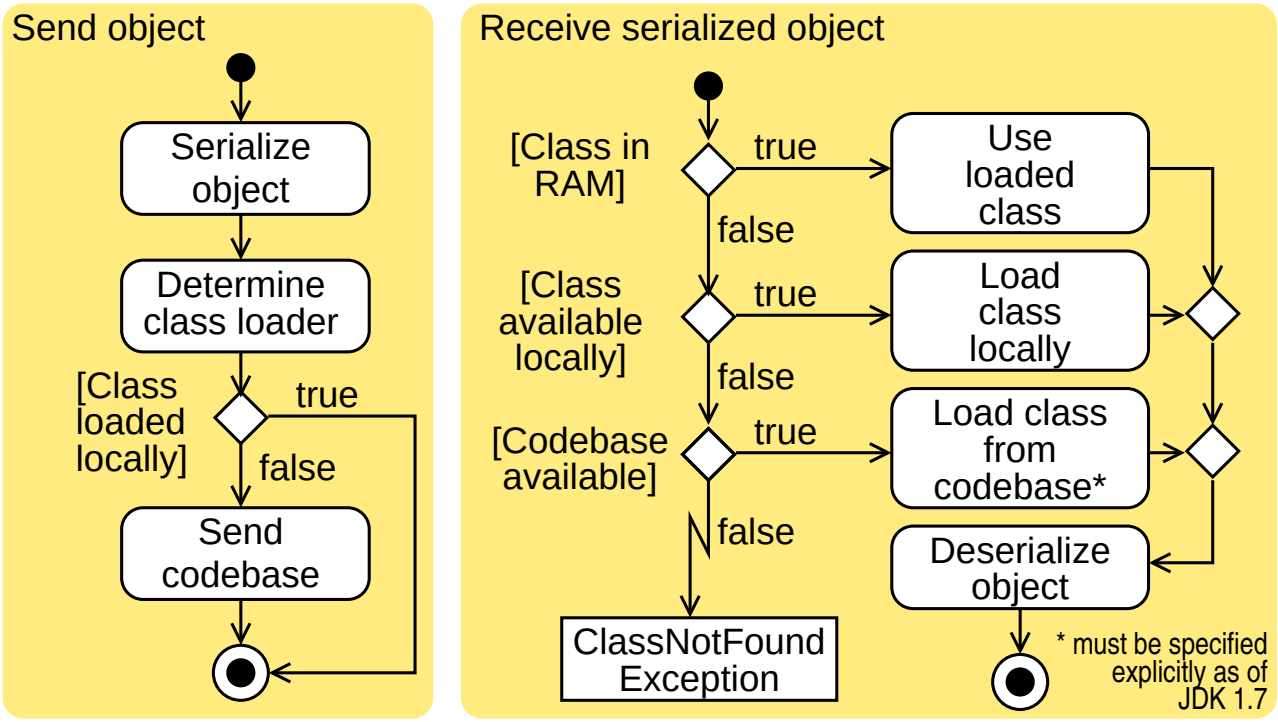
As of JDK 7 (Update 21), the default behavior for the codebase has been changed. Up to this version, the codebase specified during the transfer of the serialized object was used for remote loading. In the newer versions, only the locally specified codebase is used. Therefore, the RMI registry and the client must be started as follows:

- `rmiregistry -J-Djava.rmi.server.codebase="http://www.bsvs.de/jars/HelloServer.jar"`
- `java -Djava.rmi.server.codebase="http://www.bsvs.de/jars/HelloServer.jar" -Djava.security.policy=policy HelloClient`

Alternatively, the behavior can be changed to the old default (not recommended):

- `rmiregistry -J-Djava.rmi.server.useCodebaseOnly=false`
- `java -Djava.rmi.server.useCodebaseOnly=false -Djava.security.policy=policy HelloClient`

Procedure for Transferring Objects



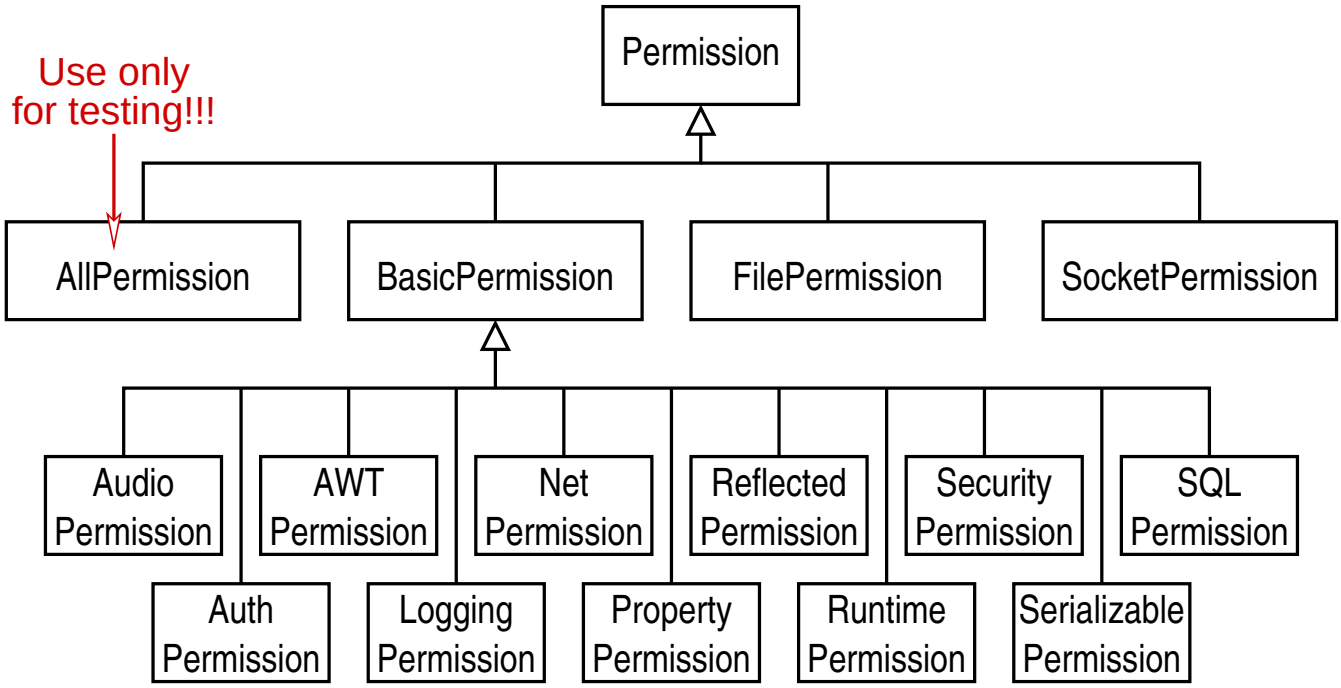
3.4.2 Java Security Manager

- JVM can be equipped with a security manager if required
  - for Java applications: `System.setSecurityManager()`
  - for Java applets: by default
- Security manager checks, among other things, whether the application is allowed to
  - access a local file,
  - establish a network connection,
  - stop the JVM,
  - create a class loader,
  - read AWT events, ...
- Permissions are specified in a security policy
  - if the specifications are violated: exception

Security Policy

- Assigns permissions to codes from specific sources
- Code source can be described by two properties:
  - code location: URL where the code was loaded from
  - certificates (for signed code)
- Permissions allow access to certain resources
  - permissions are modeled by objects, but are usually specified in the policy file
  - e.g. `FilePermission p = new FilePermission("/tmp/*", "read,write");`
  - or `permission java.io.FilePermission "/tmp/*", "read,write";`

Hierarchy of Permission Classes (JDK 1.2)



3.4.2 Java Security Manager ...

Policy File

```
grant {
  permission java.net.SocketPermission "www.bsvs.de:80",
    "connect";
};

grant codebase "file:" {
  permission java.io.FilePermission "/home/tom/-",
    "read, write";
  permission java.io.FilePermission "/bin/*", "execute";
};

grant codebase "http://www.bsvs.de/jars/HelloServer.jar" {
  permission java.net.SocketPermission "localhost:1024-",
    "listen, accept, connect";
};
```


3.4.2 Java Security Manager ...

Policy File ...

- All classes are allowed to:
  - establish connections to `www.bsvs.de`, port 80
- Locally loaded classes may:
  - read and write files in `/home/tom` or (recursively) a subdirectory of it
  - execute files in the `/bin` directory
- Classes loaded from `http://www.bsvs.de/jars/HelloServer.jar` are allowed to:
  - accept / establish network connections on / to the local computer via non-privileged ports (1024 or higher)

Further Documentation

- General information on policy files::
  http://docs.oracle.com/javase/8/docs/technotes/guides/security/PolicyFiles.html
- Overview of the permission classes::
  http://docs.oracle.com/javase/8/docs/technotes/guides/security/permissions.html
- Java API documentation::
  http://docs.oracle.com/javase/8/docs/api/

3.5 Summary

- RMI allows access to remote objects
  - transparent, via proxy objects
  - proxy classes are generated automatically
    - usually at runtime
- Parameter passing semantics
  - *by value*, if parameter object can be serialized
  - *by reference*, if parameter object is an RMI object
- Classes can also be loaded remotely (security manager!)
- Name service: RMI registry
- Security: RMI over SSL is possible, but not ideal