# Java RMI



## Properties of RMI

Remote Method Invocation (RMI) provides a framework to call methods on objects that can reside on different machines

Provide tight integration with Java

- no other language supported
- work in homogeneous environment

All remote objects have to have a remote interface

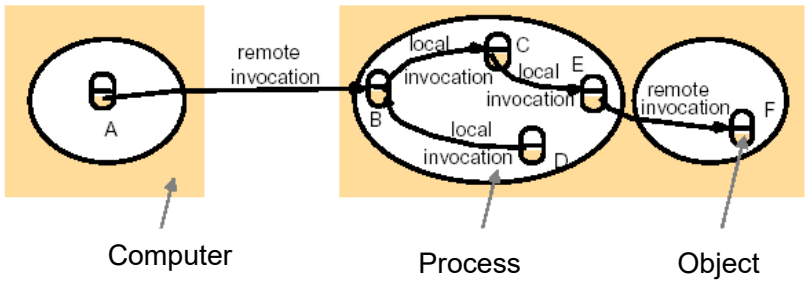
Tools for the generation of necessary classes

Nameservice





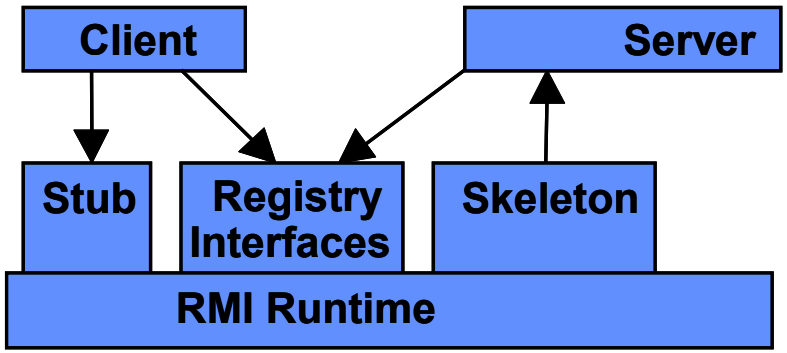
## Remote and Local Method Calls








## Java RMI Architecture



# Remote Interfaces

A remote interface describes how a remote object is accessed

Includes

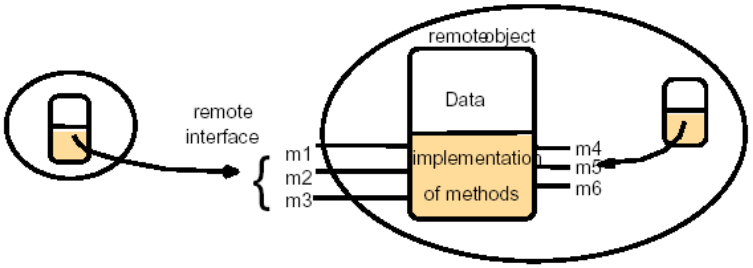
- name of interface
- possibly definitions of data types
- signature of all methods available remotely
  - method name
  - parameter
  - return type

Each middleware includes a language to define such interfaces: an Interface Definition Language (IDL)





# Remote Object and its Interface








# Remote Interfaces in Java RMI

Definition requires

- Use of Java `interface` construct
- Derived from interface `Remote`
- Methods have to throw `RemoteExceptions`

All Java Types can be used

Newly defined classes can be used

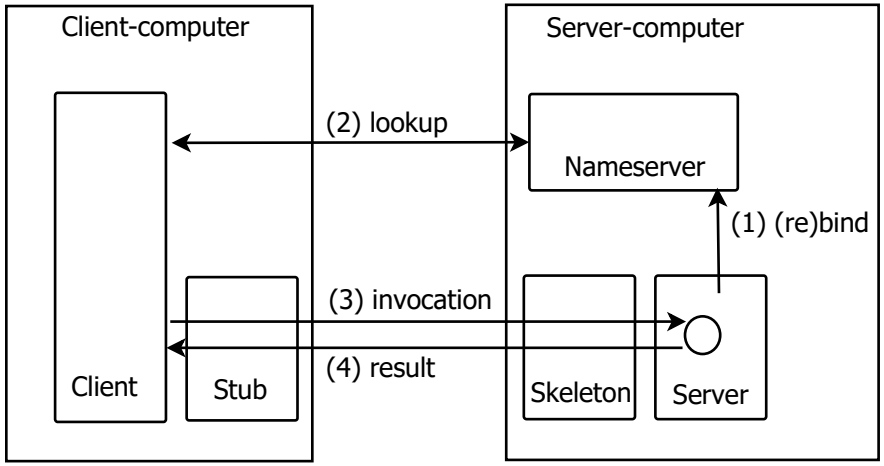




# The Interface

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

public interface DateServer extends Remote {

  public Date getDate () throws RemoteException;

}
```

# Java RMI Method Execution








# Development of a RMI Server

The server has two parts

- Servant
  - Implementation of a class for remote objects implementing a remote interface
    - `implements` <name of Interface>
  - UnicastRemoteObject as base class
    - non-replicated server
    - communication via TCP
    - lifetime of objects limited to lifetime of process that creates object
  - Constituents
    - Constructor without parameters for remote object
    - implementation of methods declared in interface
    - possibly implementation of additional methods
- "real" Server
  - Constituents
    - Maybe creation of a SecurityManager
    - Creation of one or more instances of remote objects (servants)
    - Registration of at least one remote object with name service





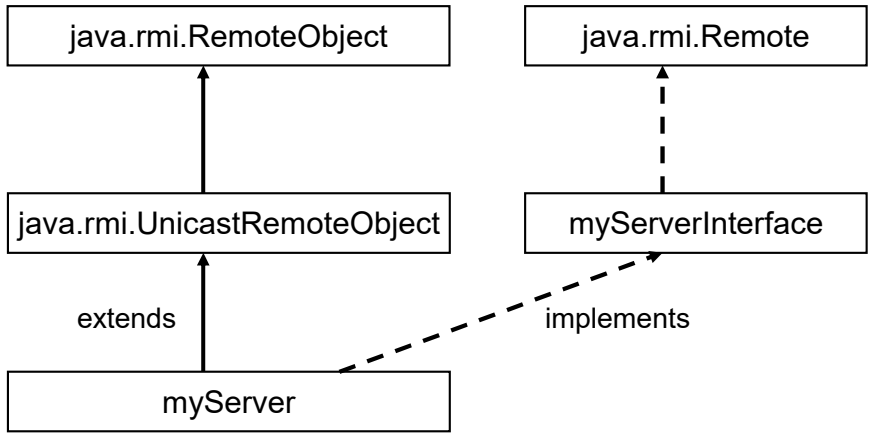
# The Server

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

public class DateServerImpl extends UnicastRemoteObject implements DateServer {

  public DateServerImpl () throws RemoteException {
  }

  public Date getDate () throws RemoteException {
    System.out.println("Call of getDate()");
    return new Date ();
  }

  public static void main (String[] args) {

    try {
      DateServerImpl dateServer = new DateServerImpl ();
      Naming.rebind ("myLittleObject", dateServer);
      System.out.println("Server started");

    } catch (Exception e) {
                          System.out.println("DateServerImpl: " + e.getMessage());
                          e.printStackTrace();
    }
  }
}
```





# RMI Classes und Interfaces



# Development of RMI Clients

Constituents

- Maybe creation of a SecurityManager
- Lookup initial object reference at the name service
- Calling methods on this object
  - these method calls can provide the client with further references to remote objects
- Possibly further method calls on other local or remote objects





# The Client

```
import java.rmi.Naming;
import java.util.Date;

public class DateClient {

  public static void main (String[] args) throws Exception {
    if (args.length != 1)
      throw new IllegalArgumentException ("Syntax: DateClient <hostname>");

    try {

      DateServer dateServer = (DateServer) Naming.lookup
          ("rmi://" + args[0] + "/myLittleObject");
      Date when = dateServer.getDate ();
      System.out.println ("Datum: " + when);

    } catch (Exception e) {
          System.out.println("DateClient: " + e.getMessage());
          e.printStackTrace();
    }

  }
}
```





# Development of a RMI-Application

1) Define Remote Interface
2) Implement Server
3) Implement Client





# RMI at Runtime

1) Start nameserver (on each server!)
   - `rmiregistry`
   - the name server has to have the remote interfaces in its ClassPath
2) Start Server
   - e.g. `java` DateServerImpl
3) Start Client
   - `java` DateClient localhost

# What Data is Exchanged?

What does a client get, when a method called on a remote object returns a result of an object-type?

- Copy of object or remote reference?
- Rule:
  - for remote objects: reference
  - for all other objects: Copy
    - This behavior is independent from the fact whether the object belongs to a predefined Java-class or to a newly defined class
    - If the class is not available at the client, the class-file itself is transferred, too.





# Nameservice

The name service for Java RMI is realized by the program `rmiregistry`

The Java RMI name service knows only the objects on its own server

- has to run on each server!

A server registers remote objects by binding them to a name

Clients request from the name service a reference to a remote object

- Query in form of an URL-like notation
- e.g.: "rmi://localhost/DateServer"





# Important Methods of java.rmi.Naming

- void rebind(String name, Remote obj)
- void bind(String name, Remote obj)
- void unbind(String name)
- Remote lookup(String name)



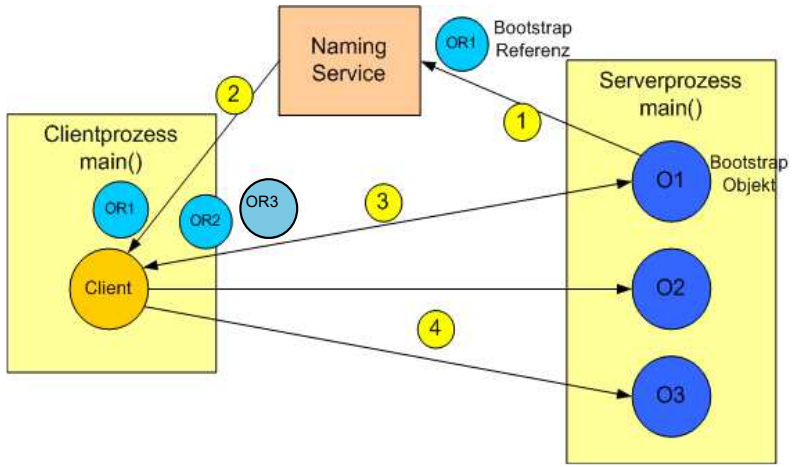


# Using the Name Service

The name service is a so called Bootstrap Service: It allows clients to locate resources to start the application

Usually, distributed systems use several servant objects. Only one (or few) of these objects are registered with the name service. These are the Bootstrap objects.

# The Bootstrap Mechanism








# Dynamic Loading of Classes

Java can load classes dynamically

- When do we need this feature?
  - When we copy an object from one process to another, the destination process might not know the class of the object
  - To use the object (e.g., call methods on the object) we need the class of the object
  - In this case we can load the class dynamically using the network
  - To load the class, we need the information where to find the class (codebase) and a mechanisms to deliver the class (webserver)
  - The objects are annotated with the codebase
  - In addition to this we, need a SecurityManager to control the allowed actions of a dynamically loaded class





# In which cases is a class missing?

A process gets an object without having access to the class of the object

Possible Cases: A parameter or return value is declared as

- Interface and the process gets an object of a class that implements this interface
- Class and the process gets an object of a sub-class





# Example

```
public class DateExtra extends Date {
  boolean geburtstag;

  public DateExtra () {
    geburtstag = false;
  }

  public DateExtra (boolean g) {
    geburtstag = g;
  }
  public String toString() {
    return super.toString() + " Geburtstag: " + geburtstag;
  }
}
```

# The Server

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

public class DateServerImpl extends UnicastRemoteObject implements DateServer {

  public DateServerImpl () throws RemoteException {
  }

  public Date getDate () throws RemoteException {
    System.out.println("Aufruf von getDate()");
    return new DateExtra ();
}

  public static void main (String[] args) {

    try {
      DateServerImpl dateServer = new DateServerImpl ();
      Naming.rebind ("meinKleinesObjekt", dateServer);
      System.out.println("Server gestartet");

    } catch (Exception e) {
                              System.out.println("DateServerImpl: " + e.getMessage());
                              e.printStackTrace();
    }
  }
}
```





# The Client

```
import java.rmi.Naming;
import java.util.Date;

public class DateClient {

  public static void main (String[] args) throws Exception {
    if (args.length != 1)
      throw new IllegalArgumentException ("Syntax: DateClient <hostname>");

    try {
     if (System.getSecurityManager() == null) {
             System.setSecurityManager(new RMISecurityManager());
     }
      DateServer dateServer = (DateServer) Naming.lookup
          ("rmi://" + args[0] + "/meinKleinesObjekt");
      Date when = dateServer.getDate ();
      System.out.println ("Datum: " + when);

    } catch (Exception e) {
          System.out.println("DateClient: " + e.getMessage());
          e.printStackTrace();
    }

  }
}
```





# Security Policies

Example of file Sicherheit.txt

```
grant {
  // allow everything
  permission java.security.AllPermission;
};
```





# Summary (1)

## Servant

- Servant is implementing methods to be accessible from remote systems
- Can also have methods that are only locally available
- Has a fixed location

## Client

- Has to know the location of the servant
- Has to know the name of the servant
- Has to know the methods of the interface

# Summary (2)

RMI allows almost homogeneous treatment of local and remote objects

- Simplifies programming
- Remaining differences
  - Initial object reference has to be obtained by querying the name service
  - Additional errors and corresponding exceptions

Implementation

- Client knows the interface of remote objects
- Remote object implements the interface

Transparency

- Access transparency
- only limited location transparency