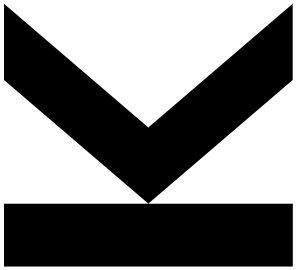

JOHANNES KEPLER
UNIVERSITY LINZ

REMOTE METHOD INVOCATION (RMI)



PR SW2 S18
Dr. Prähofer
DI Leopoldseder

OUTLINE

1. Motivation
2. Architecture
3. Remote Objects
4. Parameter Handling
5. Callbacks
6. RMI & Threads
7. (Distributed) Garbage Collection (GC)
8. Distribution and Class Loading
9. Literature

OUTLINE

1. **Motivation**
2. Architecture
3. Remote Objects
4. Parameter Handling
5. Callbacks
6. RMI & Threads
7. (Distributed) Garbage Collection (GC)
8. Distribution and Class Loading
9. Literature

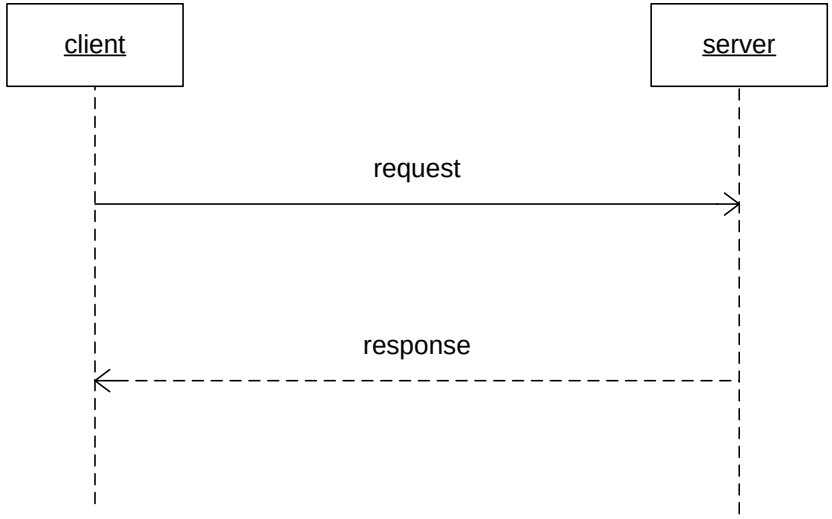
IDEA

- Do not care about **where an object resides**
- Transparent client / server boundaries
- Method calls across VM boundaries
  - Client and server **interact** on the (logically) **same object**

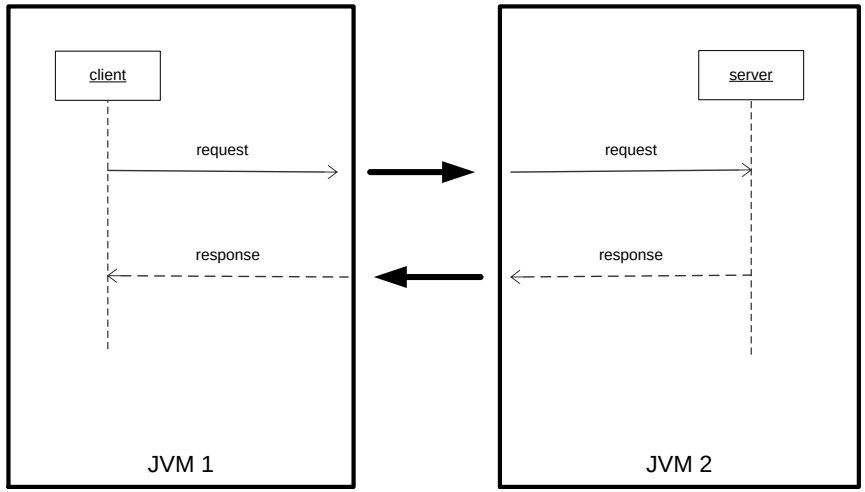
Code Demo: Hello World

MOTIVATION – DISTRIBUTED SYSTEMS

- Normal Method Call



- Remove Method Call
  - Calls between **different VMs**
  - Virtual calls require **references** → can be **remote**



https://docs.oracle.com/javase/7/docs/technotes/guides/rmi/hello/hello-world.html

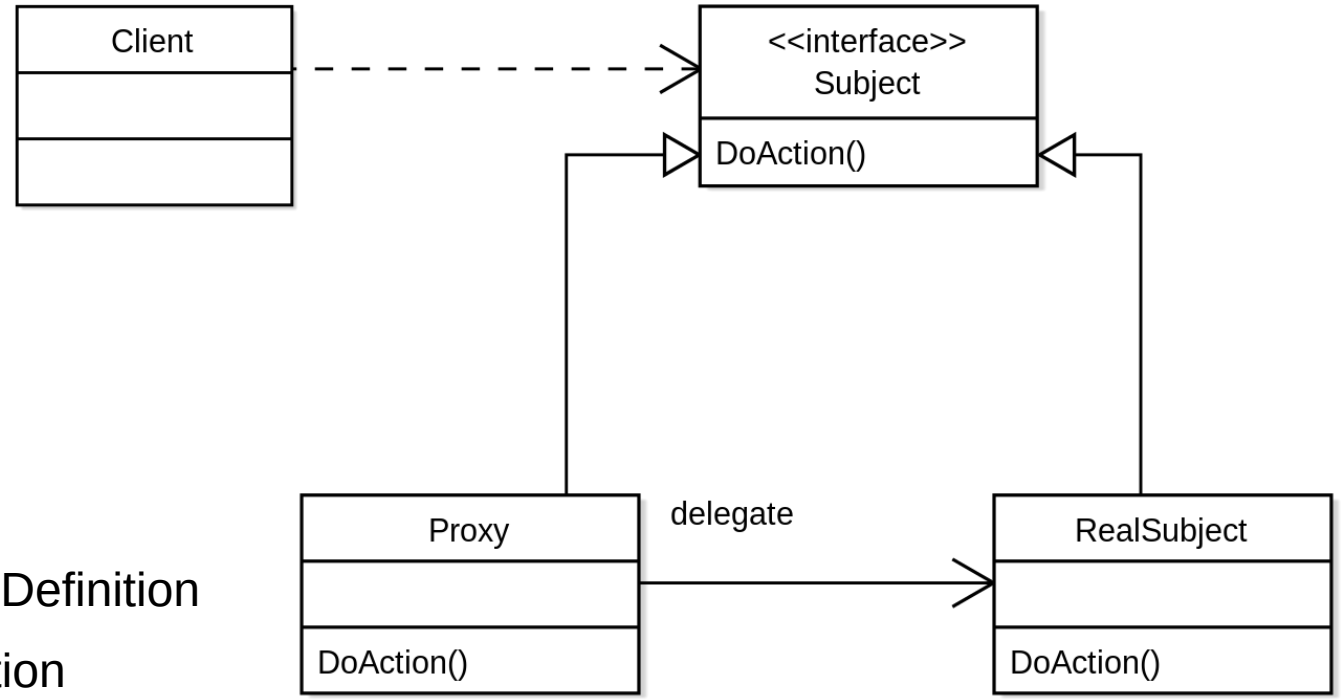
CHALLENGES REMOTE METHOD INVOCATION

- Object References
  - What is a remote reference ?
  - How to handle (the semantics of) an object reference in a different JVM? (Different VM means different heap!!)
- Method Calls
  - How to know which method is actually called ? (Virtual dispatch!)
- Parameter / Return Value Handling
  - By Value / By Reference ?
- Object Creation
  - Where to allocate an object? (Locally vs. remote?)
- Garbage Collection
  - How to handle GC roots over VM boundaries?
- Class Loading
  - When/ how / who loading the class of a remote object ?

OUTLINE

1. Motivation
2. **Architecture**
3. Remote Objects
4. Parameter Handling
5. Callbacks
6. RMI & Threads
7. (Distributed) Garbage Collection (GC)
8. Distribution and Class Loading
9. Literature

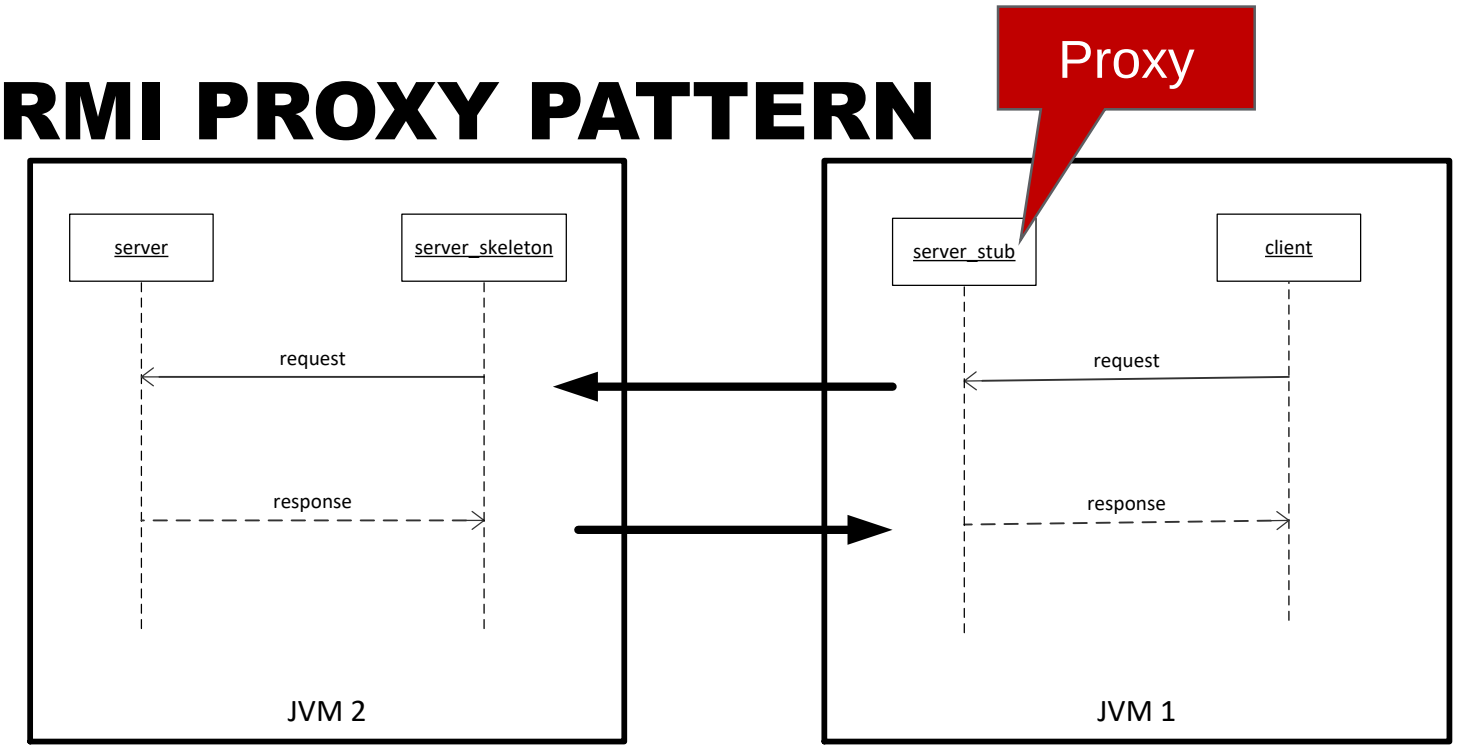
REMOTE OBJECT IMPLEMENTATION VIA PROXY PATTERN



- Interface
  - Contract Definition
- Implementation
  - Provides Functionality
- Proxy
  - Used by Client
  - Delegates requests to concrete implementation (where ever that is)

JKU

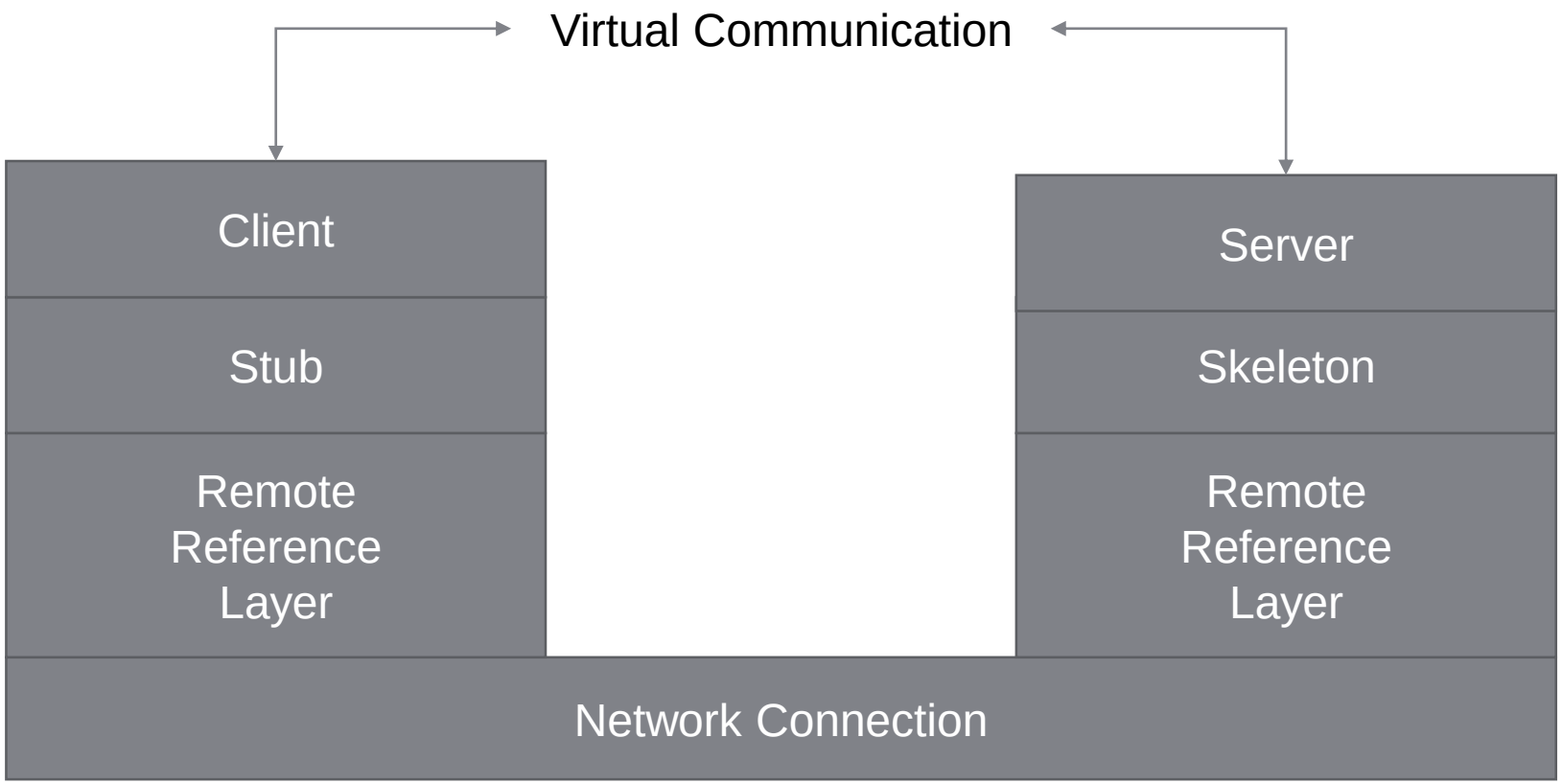
RMI PROXY PATTERN



- Stub acts as **proxy** for the server object; client(s) use the proxy stub which re-directs to server skeleton
- Proxy handles network connection to server
- Skeleton incoming point for all requests from proxy to server object
- Skeleton forwards to real implementation
- Implementation performs action
- Results sent back to client from skeleton over proxy

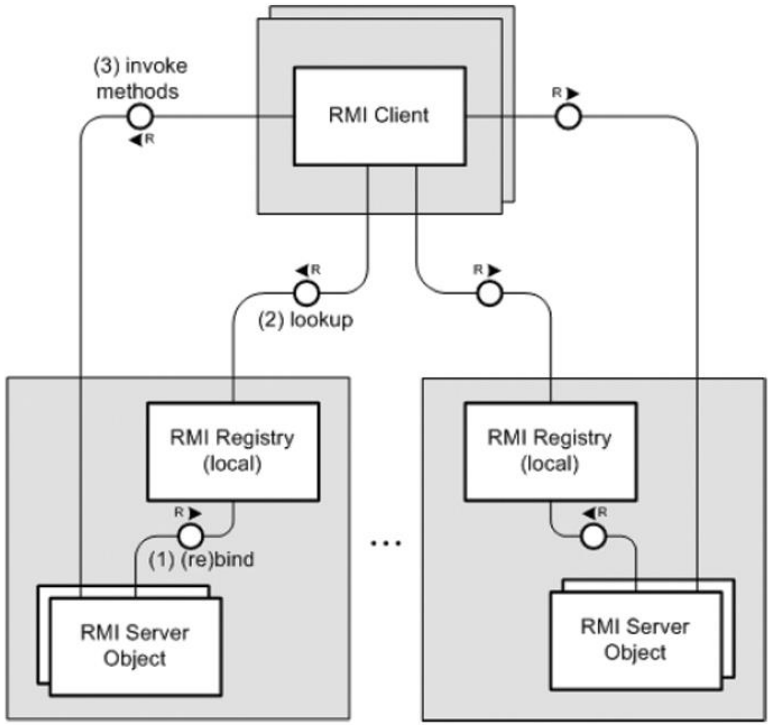
JKU

COMMUNICATION ARCHITECTURE



HOW TO FIND REMOTE OBJECTS ? – REMOTE OBJECT REGISTRATION

- One Registry per server
  - Register remote objects with RMI-URL
  - Lookup of remote objects
- Provided by
  - RMI Registration service utility `rmiregistry`
  - RMI service classes `Naming`, `Registry` and `LocateRegistry`



REGISTRY - LOOKUP

- Server

```
try {
 Bank bank = new BankImpl();
 Registry reg = LocateRegistry.createRegistry(port);
 reg.bind("Bank", bank);
}
...
```

Port

Unique Name

  - Register Object with unique RMI URL
    `rmi://<HostComputer>:1099/Bank`

- Client

```
// Client
try {
 Registry reg = LocateRegistry.getRegistry("<Hostcomputer>", port);
 Bank bank = (Bank) reg.lookup("Bank");
 ...
}
...
```

Server and Port

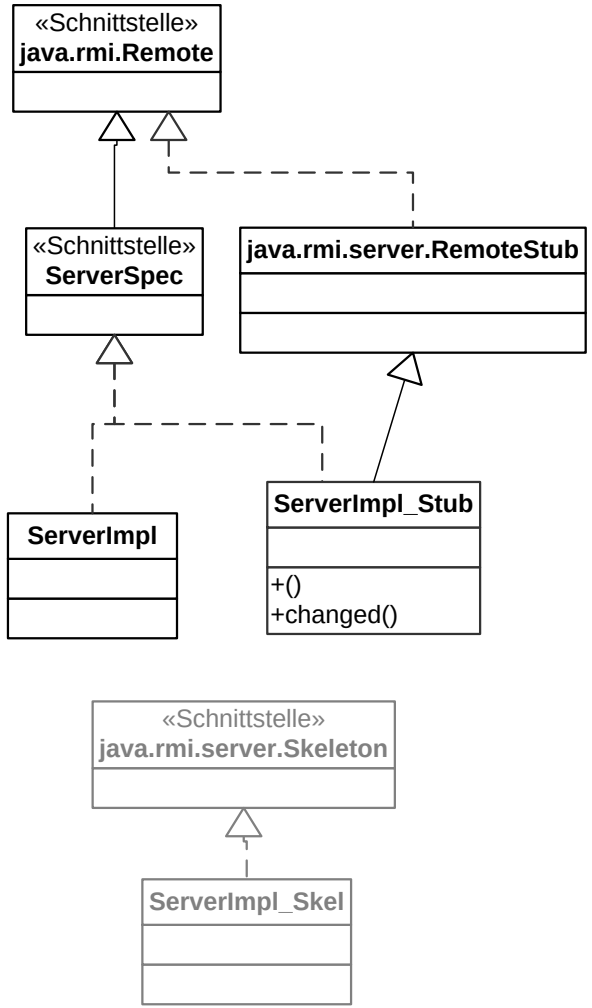
Unique Name

OUTLINE

1. Motivation
2. Architecture
3. **Remote Objects**
4. Parameter Handling
5. Callbacks
6. RMI & Threads
7. (Distributed) Garbage Collection (GC)
8. Distribution and Class Loading
9. Literature

REMOTE OBJECTS

- `java.rmi.*` defines a set of classes and interfaces
- Interfaces must extend `java.rmi.Remote`
  - Contract for public remote object methods (must throw `RemoteException`)
- Server Object Implementation
  - Implement `Remote` interface and export via `Registry`
- Stubs
  - Implements Interface
  - Since Java 1.5 auto generated
- Skeletons
  - Since Java 1.5 auto generated



JKU

SAMPLE APPLICATION – BANK SERVER

- Workflow RMI Application Development
  1. Define Remote Object Contracts (Behavior) in interfaces
  2. Implement interfaces for server
  3. Implement Server Main creating and exporting remote object(s)
  4. Implement client which looks up remote object
- Workflow Application Execution
  1. Start Server
  2. Start Client(s)

Code Demo: Simple Bank

BANK SERVER: INTERFACE BANK

- Contract

```
package bank.common;

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

public interface Bank extends Remote {

   public static final int PORT = 1099;

   public long getBalance(int account)
         throws RemoteException;

   public void deposit(int account, long amount)
         throws RemoteException;

   public boolean transfer(int from, int to, long amount)
         throws RemoteException;

}
```

RemoteExceptions required

SERVER IMPLEMENTATION

```
package bank.server;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;
import bank.common.Bank;

public class BankImpl extends UnicastRemoteObject implements Bank {
   private long[] accounts = new long[100];

   protected BankImpl() throws RemoteException { super(); }

   public synchronized long getBalance(int account) throws RemoteException {
      return accounts[account];
   }

   public synchronized void deposit(int account, long amount)
         throws RemoteException {
      accounts[account] += amount;
   }

   public synchronized boolean transfer(int from, int to, long amount)
         throws RemoteException {
      accounts[from] -= amount;
      accounts[to] += amount;
      return true;
   }
}
```

UnicastRemoteObject

Ctor
RemoteException

BANK SERVER

```
package bank.server;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import bank.common.Bank;

public class BankServer {
   public BankServer() {
      try {
         Bank bank = new BankImpl();
         Registry reg = LocateRegistry.createRegistry(Bank.PORT);
         reg.bind("Bank", bank);
      } catch (Exception e) {
         System.out.println("Trouble: " + e);
      }
   }

   public static void main(String args[]) {
      new BankServer();
   }
}
```

Allocate Remote Object

Export to Registry

BANK CLIENT

```
package bank.client;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import bank.common.Bank;

public class BankClient {

   public static void main(String[] args) {
      try {
         Registry reg = LocateRegistry.getRegistry("…Server Adr…", Bank.PORT);
         Bank bank = (Bank) reg.lookup("Bank");
         bank.deposit(1, 10000);
         bank.deposit(2, 20000);
         boolean success = bank.transfer(1, 2, 3000);
         if (success) {
            System.out.println(bank.getBalance(1));
            System.out.println(bank.getBalance(2));
         } else { … }
      } catch (Exception e) {
         …
      }
   }
}
```

Server Name

Get Remote Registry

Get Remote Object

EXPORT REMOTE OBJECTS

- Objects extending `UnicastRemoteObject` are exported explicitly in the `super()` ctor
  - Preferred way as `UnicastRemoteObject` defines
    - Remote object semantics for `equals()`
    - Remote object semantics for `hashCode()`
    - Remote object semantics for `clone()`
- Static method UnicastRemoteObject.exportObject

```
public class BankImpl implements Bank { ... }

public class BankServer {
   private Bank bank;
   public BankServer() {
      try {
         bank = new BankImpl();
         Registry reg = LocateRegistry.createRegistry(Registry.REGISTRY_PORT);
         Remote rstub = UnicastRemoteObject.exportObject(bank, Bank.PORT);
         reg.bind("Bank", bank);


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

UNEXPORT REMOTE OBJECTS

- Remove Remote Objects from RMI Registry (of server)

```
public static boolean unexportObject(Remote obj, boolean force)
        throws java.rmi.NoSuchObjectException
```

```
public class BankServer {
   private Bank bank;
   public start() {
      try {
         bank = new BankImpl();
         Registry reg = LocateRegistry.createRegistry(PORT);
         RemoteStub bstub = UnicastRemoteObject.export(bank);
         reg.bind("Bank", bstub);

      } catch (Exception e) {
         System.out.println("Trouble: " + e);
      }
   }

   public static void main(String[] args) throws Exception {
      new BankServer().start();
      // wait for termination
      …
      UnicastRemoteObject.unexportObject(bank, true);

   }
```

RMI DYNAMIC PROXIES

- `java.lang.reflect.Proxy`
- JDKs way of defining proxy classes during runtime for any given contract (specified as list of Interfaces)

```
public class Proxy {
   public static Object newProxyInstance(ClassLoader loader,
                                          Class<?>[] interfaces,
                                          InvocationHandler h)
         throws IllegalArgumentException;

   …
}
```

- JDK creates new class based on specified interfaces that will re-direct all calls to dynamically bound interface methods to an instance (defined as parameter) of a class InvocationHandler

DYNAMIC PROXY EXAMPLE – INVOCATION HANDLER

```
class TraceHandler implements InvocationHandler {
   private Object target;
   private PrintStream traceLog;

   public TraceHandler(Object target, PrintStream traceLog) {
      this.target = target;
      this.traceLog = traceLog;
   }
   public Object invoke(Object proxy, Method m, Object[] args)
            throws Throwable {
      traceLog.print(target + "." + m.getName() + "(");
      if (args != null) {
         for (int i = 0; i < args.length; i++) {
            traceLog.print(args[i]);
            if (i < args.length - 1)
               traceLog.print(", ");
         }
      }
         traceLog.println(")");
      return m.invoke(target, args);
   }
}
```

Reflective Call of method Handle

DYNAMIC PROXY EXAMPLE – MAIN PROGRAM

```
public class ProxyTest {

   public static void main(String[] args) {
      Object[] elements = new Object[1000];
      // fill elements with proxies for the integers 1 ... 1000
      for (int i = 0; i < elements.length; i++) {
         Integer value = i + 1;
         Class[] interfaces = value.getClass().getInterfaces();
         InvocationHandler handler = new TraceHandler(value, System.out);
         Object proxy = Proxy.newProxyInstance(null, interfaces, handler);
         elements[i] = proxy;
      }

      Integer key = 547;
      // search for the key
      int result = Arrays.binarySearch(elements, key);

      // print match if found
      if (result >= 0)
         System.out.println(elements[result]);
   }
}
```

{ Comparable<Integer> }

Code Demo: DynamicProxy

OUTLINE

1. Motivation
2. Architecture
3. Remote Objects
4. **Parameter Handling**
5. Callbacks
6. RMI & Threads
7. (Distributed) Garbage Collection (GC)
8. Distribution and Class Loading
9. Literature

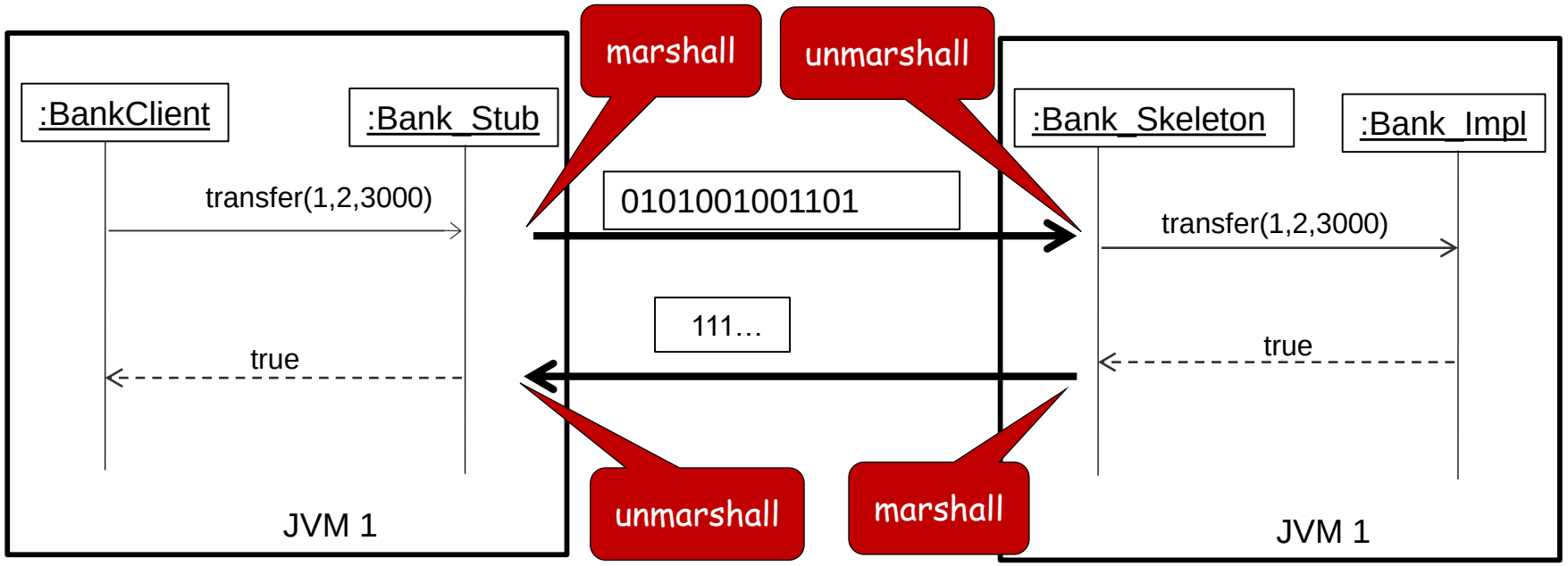
PARAMETER / RETURN VALUE HANDLING

- Parameters and return values need to be sent over the network
- What does that mean ?
  - Primitives
  - Objects
  - (Code)

- Objects are **marshalled** over the network

PARAMETER / RETURN VALUE HANDLING

```
boolean success = bank.transfer(1, 2, 3000);
```



MARSHALLING

- Primitive Data Types
  - As byte stream
- Serializable Objects
  - Are serialized, sent over and a copy is created at the receiver where the content of the object is de-serialized
- Remote objects
  - Object lives at server
  - Stub created at client side (→ Proxy)
- Not serializable
  - Cannot be used as parameters or return values

SERIALIZABLE OBJECTS

Code Demo: Serialization

- All **non-transient** fields will be serialized

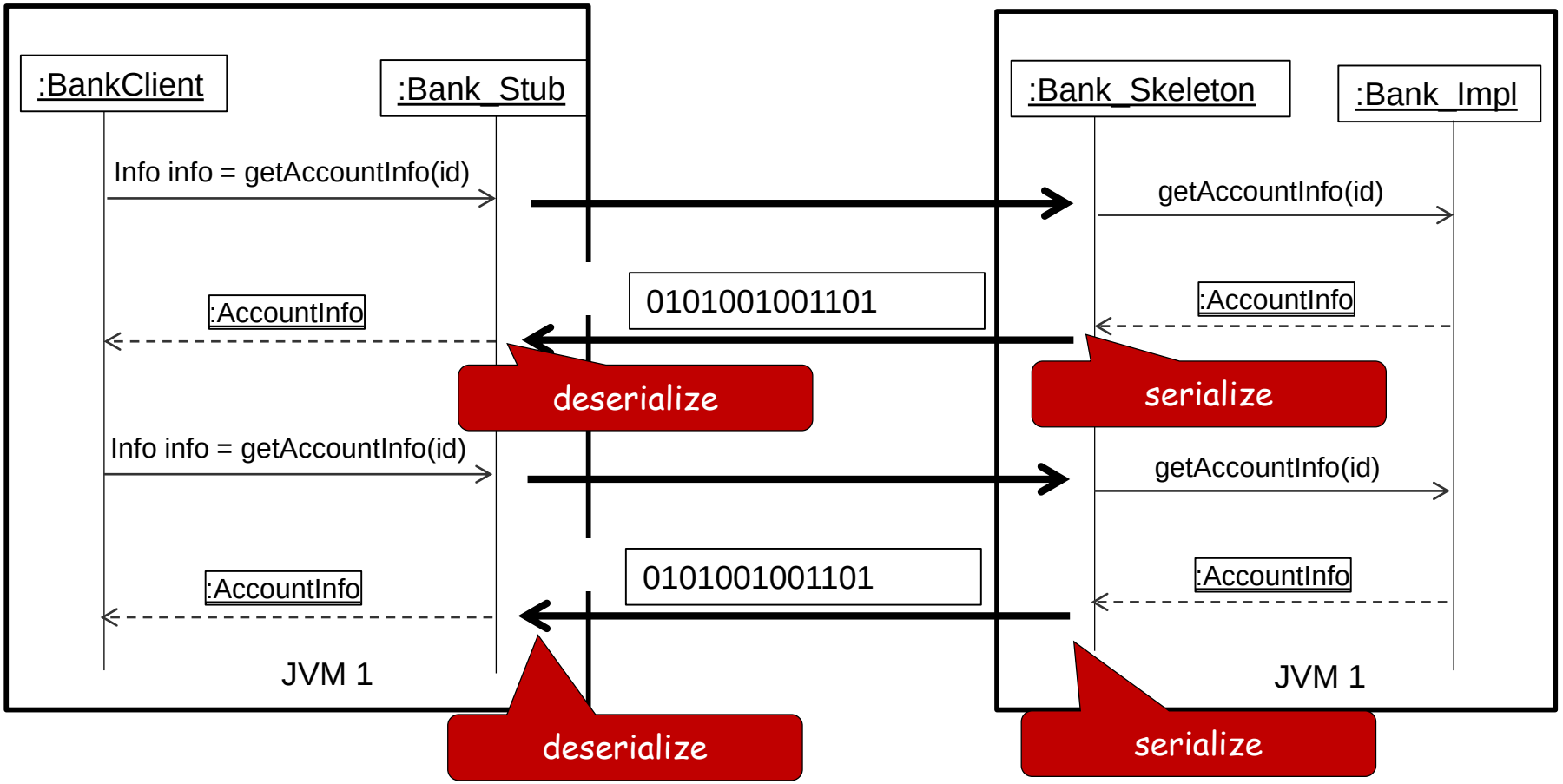
```
public class AccountInfo implements Serializable {
   private final int id;
   private final String owner;
   private final long balance;

   public AccountInfo(int id, String owner, long balance) {…}
   …
}
```

```
public interface Bank extends Remote {
   …
   public AccountInfo getAccountInfo(int accountNumber) throws RemoteException;
}
```

- Deserialization at Client

```
public class BankClient {
   public static void main(String[] args) {
      try {
         …
         AccountInfo info = bank.getAccountInfo(acc1.getId());
         …
```

BANK EXAMPLE

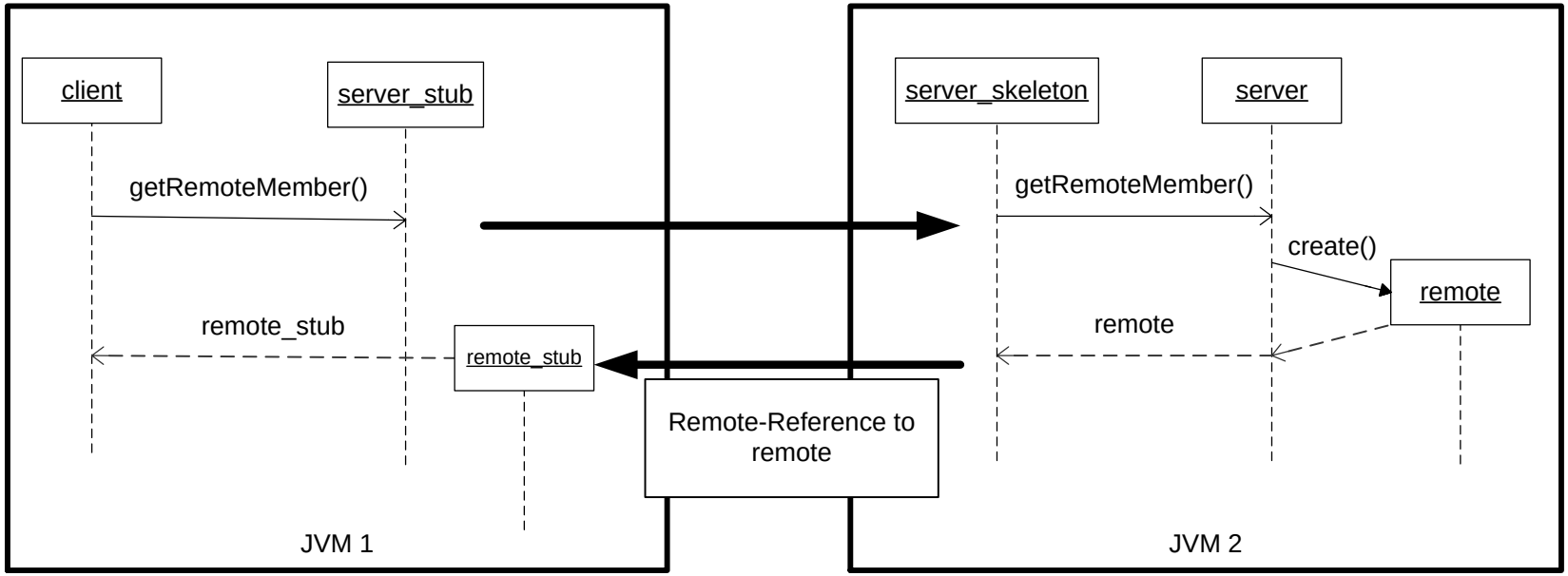


WHAT CAN BE SERIALIZED?

- Serializable
  - Everything user defined if it **implements** Serialzziable Interface
  - Primitive Data Types
  - Arrays if the **Component Type** is **serializable**
  - String
  - Collections
  - Calenders
  - Date
  - ….
- Not Serializable
  - Everything that **requires local context/semantics** to be useful
    - Threads, Class Loaders, VM Objects, OS Objects, Streams, …

REMOTE OBJECT MARSHALLING

- Remote objects can be “transmitted”
- Reference is used & stub (Proxy) is created on client side



BANK EXAMPLE - CONTRACT

```
public interface Account extends Remote {
   public int getId() throws RemoteException;
   public Customer getCustomer() throws RemoteException;
   public long getBalance() throws RemoteException;
   public long withdraw(long amount) throws RemoteException;
   public void deposit(long diff) throws RemoteException;

}
```

Remote Exceptions

```
public interface Customer extends Remote {
   public String getName() throws RemoteException;
   public Account[] getAccounts() throws RemoteException;
}
```

```
public interface Bank extends Remote {
   public Customer getCustomer(String name) throws RemoteException;
   public Customer createCustomer(String name) throws RemoteException;
   public Account getAccount(int accountNumber) throws RemoteException;
   public Account createAccount(String customer) throws RemoteException;
   public void transfer(int from, int to, long amount) throws RemoteException;
}
```

Remote Objects

JKU

SERVER SIDE – BANK IMPLEMENTATION

- **BankImpl** stores customers and accounts
- Provides access to customers and accounts

```
public class BankImpl extends UnicastRemoteObject implements Bank {
   private static BankImpl instance;
   public synchronized static Bank getInstance() {
      if (instance == null) {
         try { instance = new BankImpl();
         } catch (RemoteException e) { }
      }
      return instance;
   }

   private static int id = 0;
   private final Map<Integer, AccountImpl> accounts;
   private final Map<String, CustomerImpl> customers;

   private BankImpl() throws RemoteException {
      accounts = new HashMap<Integer, Account>();
      customers = new HashMap<String, Customer>();
   }
   @Override
   public synchronized Customer getCustomer(String name) throws RemoteException {
      return customers.get(name);
   }
   @Override
   public synchronized Account getAccount(int id) throws RemoteException {
      return accounts.get(id);
   }
   ...
```

SERVER - CUSTOMER IMPLEMENTATION

```
public class CustomerImpl extends UnicastRemoteObject implements Customer {

   private final String name;
   final List<Account> accounts;

   protected CustomerImpl(String name) throws RemoteException {
      super();
      this.name = name;
      this.accounts = new ArrayList<Account>();
   }

   @Override
   public String getName() throws RemoteException {
      return name;
   }

   @Override
   public synchronized Account[] getAccounts() throws RemoteException {
      return accounts.toArray(new Account[0]);
   }

}
```

SERVER – ACCOUNT IMPLEMENTATION

```
public class AccountImpl extends UnicastRemoteObject implements Account {

   private final int id;
   private final String customer;
   private long balance;

   public AccountImpl(int id, String customer) throws RemoteException {
      super();
      this.id = id;
      this.customer = customer;
      balance = 0L;
   }

   @Override
   public int getId() throws RemoteException {
      return id;
   }

   @Override
   public Customer getCustomer() throws RemoteException {
      return BankImpl.getInstance().getCustomer(customer);
   }

   @Override
   public synchronized long getBalance() throws RemoteException {
      return balance;
   }
   …
}
```

SERVER – MAIN PROGRAM

```
public class BankServer {

   private static Bank bank;

   private static void start() throws Exception {
      bank = BankImpl.getInstance();
      Registry reg = LocateRegistry.createRegistry(Registry.REGISTRY_PORT);
      reg.bind("Bank", bank);

      System.out.println("Server started on port " + Registry.REGISTRY_PORT);
   }

   public static void main(String[] args) throws Exception {
      start();

   …
}
```

CLIENT – MAIN PROGRAM

```
public class BankClient {
   …
   public static void main(String[] args) {
      try {
         Registry reg = LocateRegistry.getRegistry("localhost", Bank.PORT);
         Bank bank = (Bank) reg.lookup("Bank");

         Customer cust1 = bank.createCustomer("Berger");
         Customer cust2 = bank.createCustomer("Maier");

         Account acc1 = bank.createAccount("Berger");
         Account acc2 = bank.createAccount("Maier");
         acc1.deposit(10000);
         acc2.deposit(20000);
         boolean success = bank.transfer(acc1.getId(), acc2.getId(), 3000);
         if (success) {
            System.out.println(acc1.getBalance());
            System.out.println(acc2.getBalance());
         } else {
            System.out.println("Transfer not successful");
         }
      } catch (Exception exc) {
         exc.printStackTrace();
      }
   }
```

Remote References

Remote References

Unmarshalled Primitives

JKU

SERIALIZATION & REMOTE OBJECTS

- Object references can be remote objects
- Those fields will point to stubs (on the client) and real objects on the server

```
public interface Customer extends Remote {
   public Account[] getAccounts() throws RemoteException;

}
```

Serializabled Array of Remote objects

```
public class BankClient {
   public static void main(String[] args) {
      try {
         …
         Account[] accounts = customer.getAccounts();
```

Creates array of stubs (proxies)

```
public class RemoteAccountEvent extends EventObject implements Serializable {
   private final Account account;

   public RemoteAccountEvent(Account source) {
      super(source);
   }
   …
}
```

NOT A GOOD IDEA : Every object contains remote stub

REFERENCE & STRUCTURAL EQUALITY

- Reference Equality **==**
  - Every de-serialization creates a new object, they are not reference equal
  - But **structural equal (**Java **equals()**)
  - Therefore, implement **equals** and **hashCode** (done by unicast remote object already, also for serialized objects)

```
public class BankClient_Equals {
   public static void main(String[] args) {
      try {
         Customer berger = bank.getCustomer("Berger");
         String name1 = berger.getName();
         String name2 = berger.getName();

         if (name1 == name2) {
```

false

```
public class BankClient_Equals {
   public static void main(String[] args) {
      try {
         Customer berger = bank.getCustomer("Berger");
         String name1 = berger.getName();
         String name2 = berger.getName();

         if (name1.equals(name2)) {
```

true

REFERENCE & STRUCTURAL EQUALITY - STUBS

- Reference Equality **==**
  - Every de-serialization creates a new object, they are not reference equal
  - But **structural equal (**Java **equals()**)
  - Therefore, implement **equals** and **hashCode** (done by unicast remote object already, also for serialized objects)

```
public class BankClient_Equals {
   public static void main(String[] args) {
      try {
         Customer cust1 = bank.getCustomer("Berger");
         Customer cust2 = bank.getCustomer("Berger");

         if (cust1 == cust2) {
```

false

```
public class BankClient_Equals {
   public static void main(String[] args) {
      try {
         Customer cust1 = bank.getCustomer("Berger");
         Customer cust2 = bank.getCustomer("Berger");

         if (cust1.equals(cust2)) {
```

true

REFERENCE & STRUCTURAL EQUALITY - STUBS

- **CustomerImpl** implements **equals**, but that has no effect on client, as client will create a **stub (proxy)** for every remote referenced looked up on the server

```
public class CustomerImpl extends UnicastRemoteObject implements Customer {
   ...
   public synchronized boolean equals(Object obj) {
      if (obj instanceof CustomerImpl) {
         return name.equals((CustomerImpl) obj).name;
      }
      return false;
   }
}
```

Only on server

```
public class BankImpl extends UnicastRemoteObject implements Bank {
   ...
   public synchronized Customer getCustomer(String name) throws RemoteException
      return new CustomerImpl(name);
}
```

Customer with same name is created

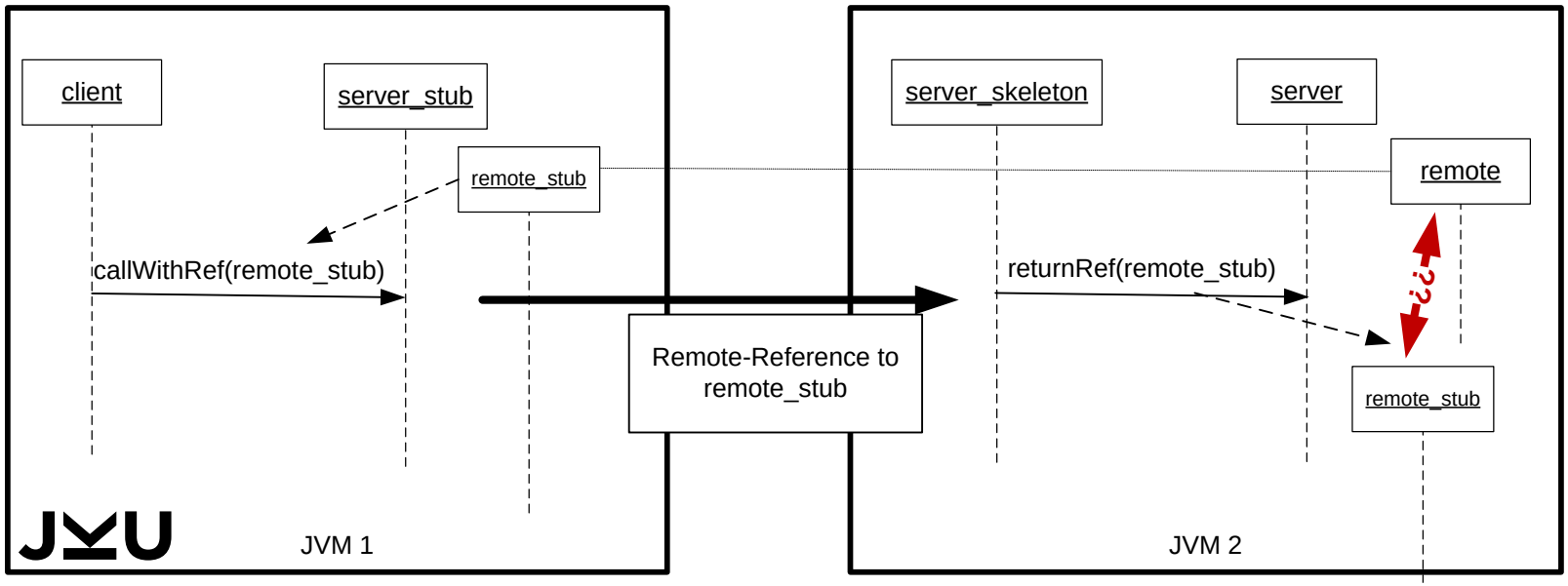
```
public class BankClient_Equals {
   public static void main(String[] args) {
      try {
         Customer berger1 = bank.getCustomer("Berger");
         Customer berger2 = bank.getCustomer("Berger");

         if (cust1.equals(cust2)) {
```

False, as equals of stub is used and they are not the same remote object

REMOTES AS PARAMETERS

```
public boolean transfer(Account from, Account to, long amount) throws RemoteException;
```

- Remote references as parameters are problematic
  - Remotes are retrieved at client
  - Client creates and holds stub object for remote reference
  - If stub is used as parameter, the stub object is sent back to the server
    - **Looses connection between original object and stub**
    - **Therefore, use ids as parameters**



NO-GOS AND BAD STYLE

```
public interface Bank extends Remote {
   …
   public boolean transfer(Account from, Account to, long amount) throws RemoteException;
}
```

- Server-sided Implementation of contract

```
public class BankImpl extends UnicastRemoteObject implements Bank {
   …
   @Override
   public boolean transfer(Account from, Account to, long amount) throws RemoteException {
      if (from == to) {
         return false; // does not work as expected!!
      }
      if (! accounts.containsValue(to)) {
         return false; // does not work as expected!!
      }

      return transfer(from.getId(), to.getId(), amount);
   }
```

Always false

Always true

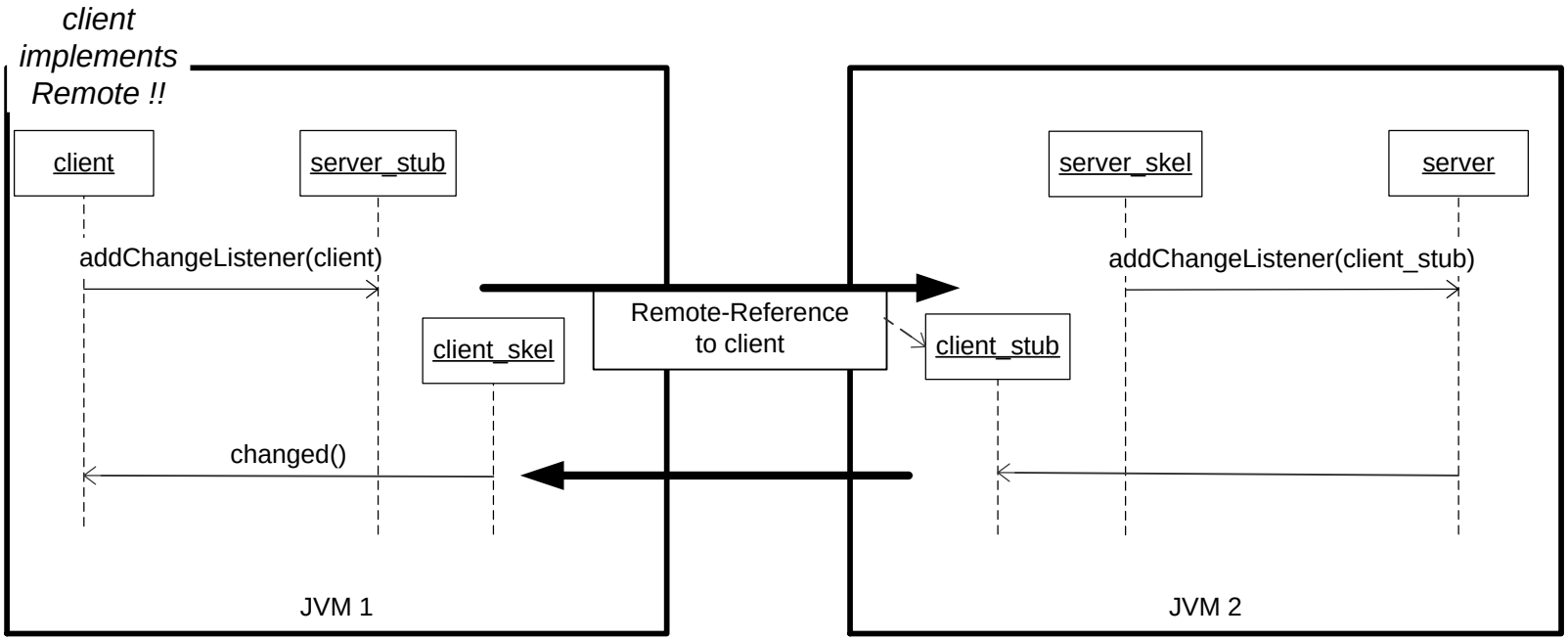
Must work with ids

OUTLINE

1. Motivation
2. Architecture
3. Remote Objects
4. Parameter Handling
5. **Callbacks**
6. RMI & Threads
7. (Distributed) Garbage Collection (GC)
8. Distribution and Class Loading
9. Literature

CALLBACKS

- Callbacks are calls **from server to client**
  - E.g. to send client a notification
- Roles of client and server are swapped
  - Client must provide a remote object
  - Server has remote reference (Proxies)



BANK WITH CALLBACKS (1)

- Remote listener for clients

```
public interface RemoteAccountListener extends Remote {
   public void accountChanged(RemoteAccountEvent e) throws RemoteException;
}
```

Client Side

- Accounts can register listeners

```
public interface Account extends Remote {
  …
  public void addRemoteAccountListener(RemoteAccountListener l)
              throws RemoteException;
  public void removeRemoteAccountListener(RemoteAccountListener l)
              throws RemoteException;
```

```
public class AccountImpl extends UnicastRemoteObject implements Account {
   private final List<RemoteAccountListener> listeners = …;
   …

   @Override
   public void addRemoteAccountListener(RemoteAccountListener l) throws RemoteException {
      listeners.add(l);
   }
   @Override
   public void removeRemoteAccountListener(RemoteAccountListener l) throws RemoteException
   listeners.remove(l);
   }
   private void fireAccountChanged() { … }
```

Server Side

BANK WITH CALLBACKS (2)

```
public class BankClient_Callback {
    public static void main(String[] args) {
        …
        try {
            …
            acc1Watcher = new AccountWatcher("Berger");
            acc2Watcher = new AccountWatcher("Maier");
            acc1.addRemoteAccountListener(acc1Watcher);
            acc2.addRemoteAccountListener(acc2Watcher);

            acc1.deposit(10000);
            acc2.deposit(20000);
            boolean success = bank.transfer(acc1.getId(), acc2.getId(), 3000);
            …
        } catch (Exception e) {
            System.out.println("Trouble: " + e);
        } finally {
            try { acc1.removeRemoteAccountListener(acc1Watcher); } catch (Exception e) {}
            try { acc2.removeRemoteAccountListener(acc2Watcher); } catch (RemoteException e) { }
            try { UnicastRemoteObject.unexportObject(acc1Watcher, true); } catch (NoSuchObjectException e) { }
            try { UnicastRemoteObject.unexportObject(acc2Watcher, true);} catch (NoSuchObjectException e) {  }

        }
    }
    private static class AccountWatcher  extends UnicastRemoteObject
                                         implements RemoteAccountListener {
        private final String name;
        public AccountWatcher(String name) throws RemoteException { … }
        public void accountChanged(RemoteAccountEvent e) throws RemoteException { … }
    }
}
```

Skeleton @ Client

Transfer remote to server

Work on server data

Server fires event and calls stubs which forwards to client

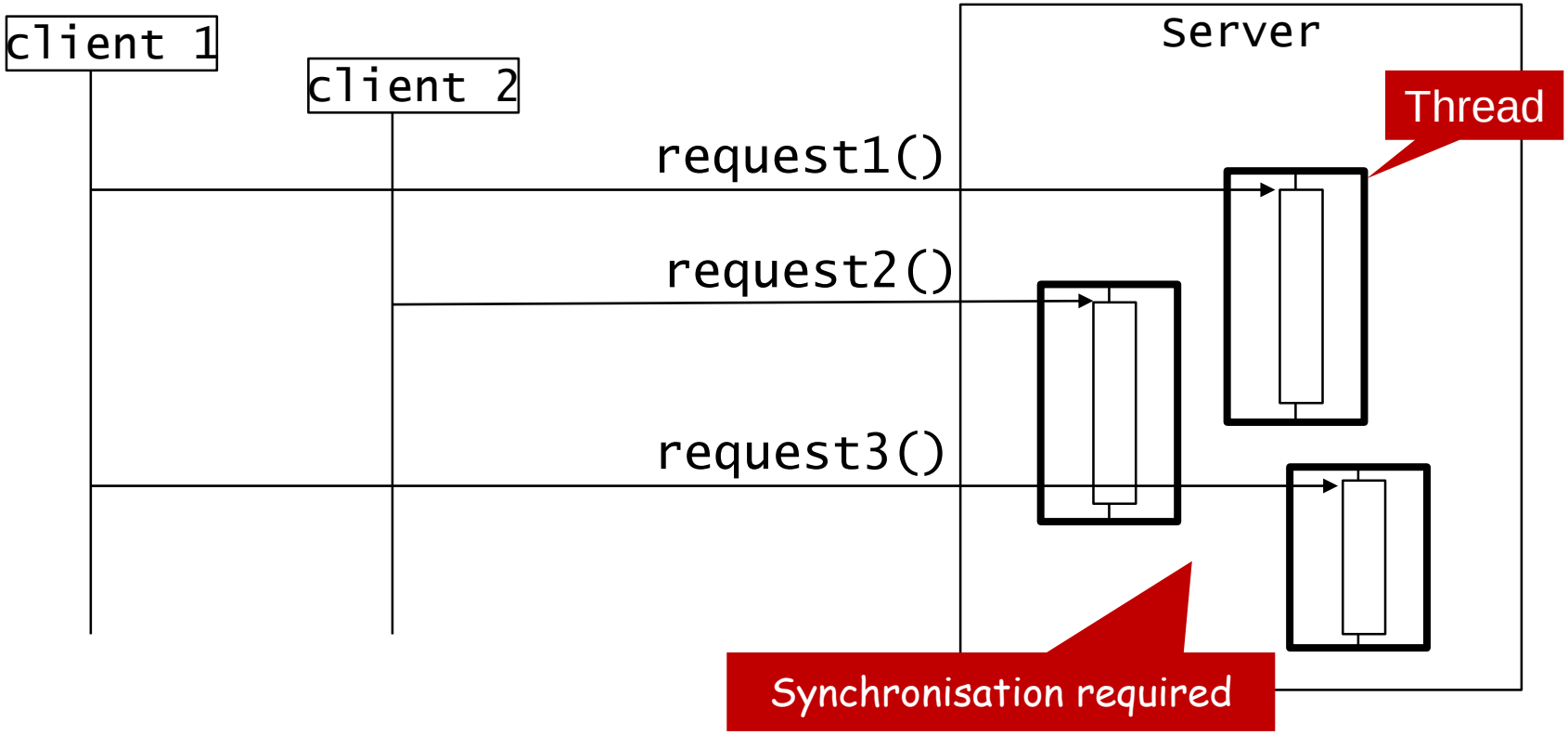
OUTLINE

1. Motivation
2. Architecture
3. Remote Objects
4. Parameter Handling
5. Callbacks
6. **RMI & Threads**
7. (Distributed) Garbage Collection (GC)
8. Distribution and Class Loading
9. Literature

JKU

RMI & THREADS

- Each request is executed in its own thread
- Therefore **synchronization required**



DIFFERENT LEVELS OF SYNCHRONIZATION

- Synchronize access to remote objects
  - Synchronize remote methods
    - **Synchronized** methods
    - **Synchronized** blocks
  - Synchronize critical regions inside remote methods
    - Monitors
    - (Reentrant) Locks
- Synchronize data
  - Server data model synchronization
  - Synchronized data structures

SYNCHRONIZE REMOTE METHODS

```
public class BankImpl extends UnicastRemoteObject implements Bank {
   ...
   @Override
   public synchronized Customer getCustomer(String name) throws RemoteException {
      return customers.get(name);
   }

   @Override
   public synchronized Customer createCustomer(String name) throws RemoteException { … }

   @Override
   public synchronized Account getAccount(int id) throws RemoteException { … }

   @Override
   public synchronized Account createAccount(String c) throws RemoteException { … }
   ...
}
```

Synchronisation on **this**

```
public class AccountImpl extends UnicastRemoteObject implements Account {
   private final List<RemoteAccountListener> listeners;
   …
   public AccountImpl(int id, String customer) throws RemoteException {
      this.listeners = new CopyOnWriteArrayList<>();
   }
   …
```

Thread-safe listener list

JKU

CHALLENGE: LONG RUNNING PROCESSES

- Long running code can block the entire application (server sided)
  - One client can block all other clients which has negative effect on entire application

```
public class BankImpl extends UnicastRemoteObject implements Bank {
   ...
   public synchronized void makeAnnualBalance() {
      // long lasting activity
      …
   }
   ...
}
```

Blocks other clients

  - Idea: Decouple long running requests from entire server via background thread

REENTRANCY

- Java locks are reentrant, i.e., thread which holds lock can enter synchronized code of same lock

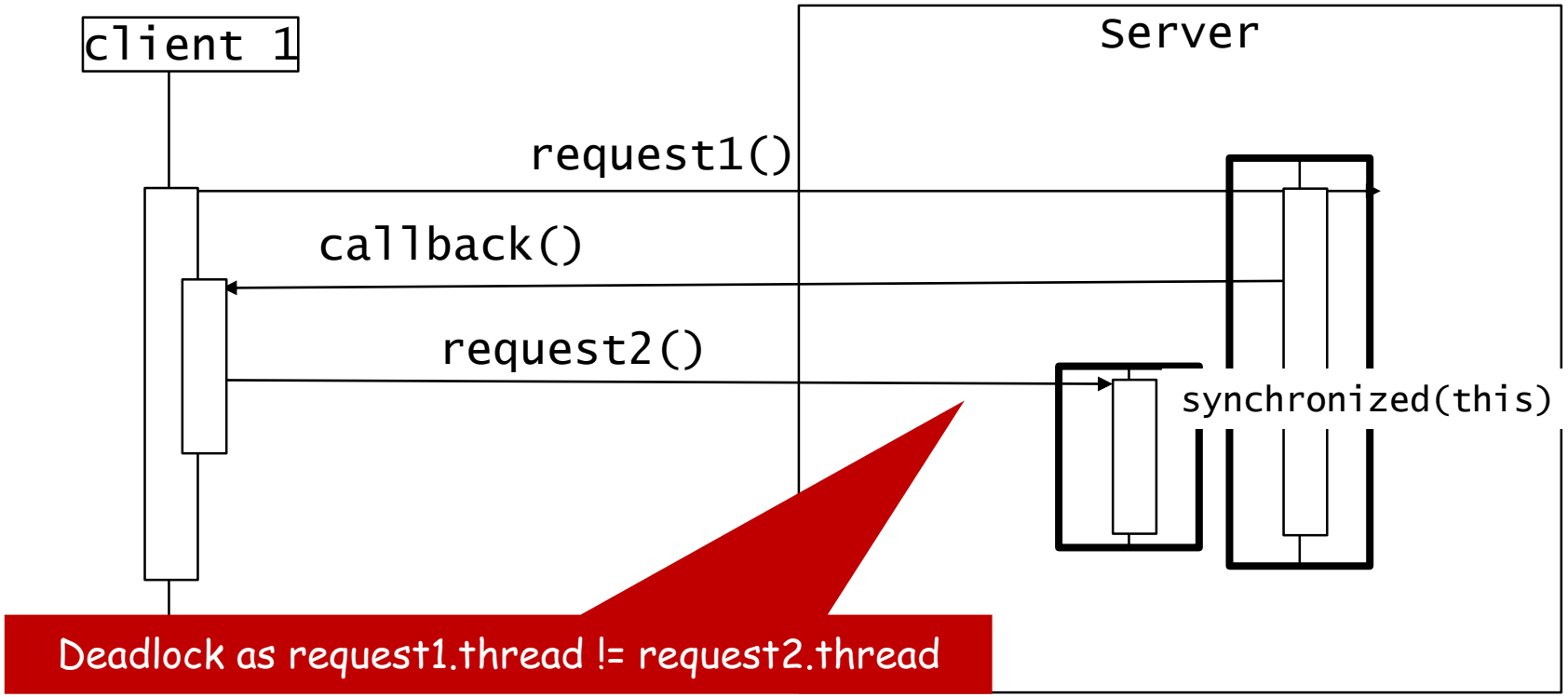
With lock to BankImpl

```
public class BankImpl extends UnicastRemoteObject implements Bank {
   ...
   @Override
   public synchronized void transfer(int from, int to, long amount) throws RemoteException {
      Account aFrom = getAccount(from);
      Account aTo = getAccount(to);
      …
   }
   @Override
   public synchronized Account getAccount(int id) throws RemoteException { … }

}
```

With lock to BankImpl → reentrant

REENTRANCY CAN CAUSE DEADLOCK

- If the server calls the client via a **callback,** and the client's implementation calls a **remote method on the server**, the thread executing the second server call is a different one than the first request to the server
  - Therefore, client's second request to server blocks as it is a different thread



DEADLOCKING CODE

Code Demo: Deadlock

- Bank Example: Remote method call in callback

```
public class BankClient extends UnicastRemoteObject {

  private static class AccountWatcher
   extends UnicastRemoteObject implements AccountListener {


      public void accountChanged(AccountChangedEvent evt)
              throws RemoteException {
         bank.getAccount()...

      }
   };
```

Client calls remote method

1: Thread a

2

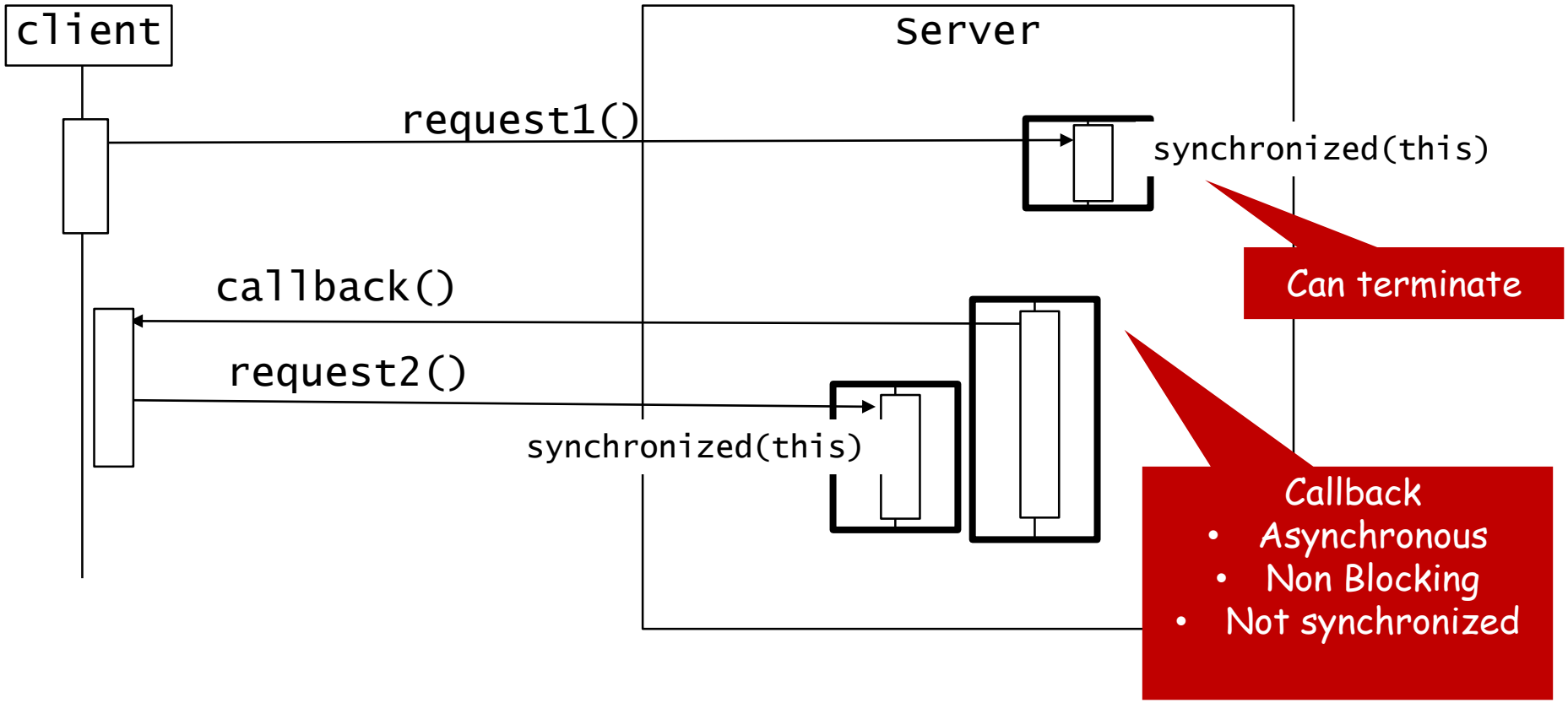
3: Thread b

Client

Server

```
public class BankImpl extends UnicastRemoteObject
   ...
   @Override
   public synchronized void transfer(int from, int to, long amount) throws RemoteException {
      Account aFrom = getAccount(from);
      Account aTo = getAccount(to);
      aTo.fireAccountChanged();

   }
   @Override
   public synchronized Account getAccount(int id) throws RemoteException { … }

}
```

SOLUTION: ASYNCHRONOUS CALL BACKS



ASYNCHRONOUS CALLBACK

```
public class AccountImpl extends UnicastRemoteObject implements Account {
   private final ExecutorService executor = Executors.newFixedThreadPool(10);
   ...
   @Override
   public void deposit(long diff) throws RemoteException {
      synchronized (this) {
         balance = balance + diff;
         fireAccountChangedEvent();
      }
   }

   private void fireAccountChanged() {
      final AccountChangedEvent evt = new AccoutChangedEvent(…);
      for (final AccountListener l : listeners) {
         executor.submit(() -> {
            l.accountChanged(evt);
         });
      }
   }
   …
}
```

Can terminate

Run callback in asynchronous Runnable

OUTLINE

1. Motivation
2. Architecture
3. Remote Objects
4. Parameter Handling
5. Callbacks
6. RMI & Threads
7. **(Distributed) Garbage Collection (GC)**
8. Distribution and Class Loading
9. Literature

JKU

DISTRIBUTED GARBAGE COLLECTION

- Normal garbage collector frees objects in VM which cannot be accessed any more (i.e. remove objects that are not needed any more, because they are not accessed)
- Cannot collect remote objects
  - No references to remote object
- Distributed Garbage Collector
  - Reference Counting: Number of references from clients to remote objects
  - Lease Time: If a client does not access a remote object for some time, it is allowed to be collected
  - **Clients need to account for the case that remote objects are no longer reachable**
    - **System property: java.rmi.dgc.leaseValue**

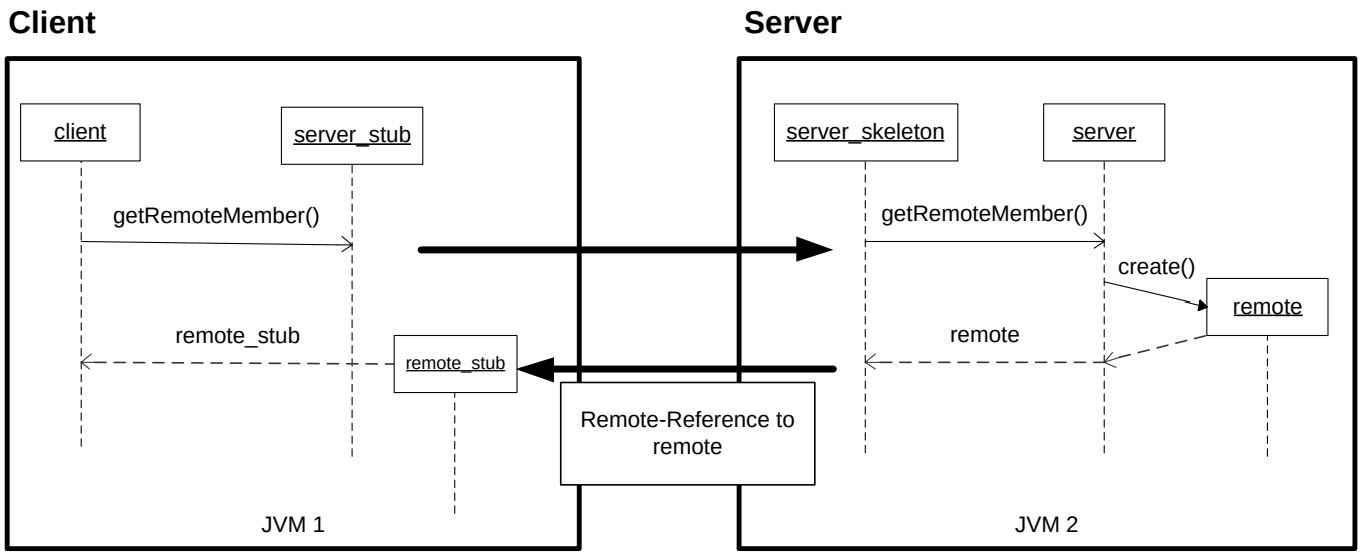
OUTLINE

1. Motivation
2. Architecture
3. Remote Objects
4. Parameter Handling
5. Callbacks
6. RMI & Threads
7. (Distributed) Garbage Collection (GC)
8. **Distribution and Class Loading**
9. Literature

JKU

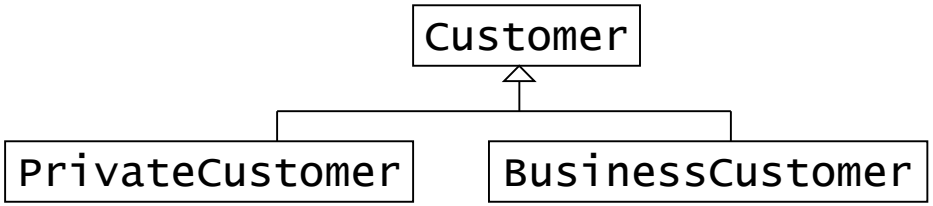
CLASS LOADING

- Clients need to dynamically load classes
  - For Stubs
  - For Parameters and return values



DYNAMIC CLASS LOADING - MOTIVATION

```
public interface Bank extends Remote {

  public Customer getCustomer(String customerName) throws RemoteException;
}
```




```
public class BankImpl extends UnicastRemoteObject implements Bank {
  public synchronized Customer getCustomer(String customerName) throws RemoteException {
      if (…) return privateClients.get(customerName);
      else return businessClients.get(customerName);
  }
```

Different sub classes from server to client

```
  try {
      Customer customer = account.getCustomer(name);
  } catch (RemoteException remoteException) {
      System.err.println(remoteExc
  }
```

Concrete objects of type PrivateCustomer_Stub or BusinessCustomer_Stub

JKU

DYNAMIC CLASS LOADING

- Code is loaded from download folder or web server
- However, in order to allow the VM to load external code and execute it permissions are needed
  - **SecurityManager** has to be installed (future lecture)
  - Program needs permissions to
    - Socket connection to RMI port
    - Socket connection for remote class loading

Example:

- **Client program:**

```
public class MyClient {
    public static void main(String[] args) {
      System.setProperty("java.security.policy", "client.policy");
      System.setSecurityManager(new SecurityManager());
      ...
```

- **Policy file for client:**

```
grant {
    permission java.net.SocketPermission "server-url:1024-65535", "connect";
    permission java.net.SocketPermission "server-url:80", "connect";
    permission java.net.SocketPermission "server-url:8080", "connect";
}
```

DISTRIBUTION OF CODE

- Code divided in 3 parts
  - Server
    - Classes required by server
  - Download
    - All classes eventually needed by client and possibly loaded by client
    - All dependent classes (hierarchy !)
  - Client
    - Classes to run the client
    - Policy file
- 3 part deployment
  - Server: Server computer
  - Download: Download folder of web server
  - Client: Computer running client application

OUTLINE

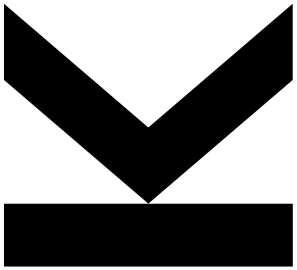
1. Motivation
2. Architecture
3. Remote Objects
4. Parameter Handling
5. Callbacks
6. RMI & Threads
7. (Distributed) Garbage Collection (GC)
8. Distribution and Class Loading
9. **Literature**

LITERATURE

- Horstmann, Cornell, Core Java 2, Volume II . Advance Features, Sun Microsystems, 2008: Chapter 5
- Krüger, Handbuch der Java-Programmierung, 3. Auflage, Addison-Wesley, 2003, http://www.javabuch.de: Kapitel 46
- Fundamentals of RMI, Short Course, http://java.sun.com/developer/onlineTraining/rmi/
- Dynamic proxies https://www.javaworld.com/article/2076234/soa/get-smart-with-proxies-and-rmi.html

JOHANNES KEPLER
UNIVERSITY LINZ

THANK YOU



JOHANNES KEPLER
UNIVERSITY LINZ
Altenberger Strasse 69
4040 Linz, Austria
www.jku.at


JOHANNES KEPLER
UNIVERSITY LINZ