Unit 4A Java Remote Method Invocation: Basics

Unit Outcomes. Here you will learn

- how to make multiple object systems (eg JVMs) work as a single system
- to program simple Java RMI applications
- how remote method invocation differs from ordinary method invocation
- how the RMI network model differs from the JMS network model and the implications these differences have for DS developers

Further Reading: CDK2005 5.1, 5.5

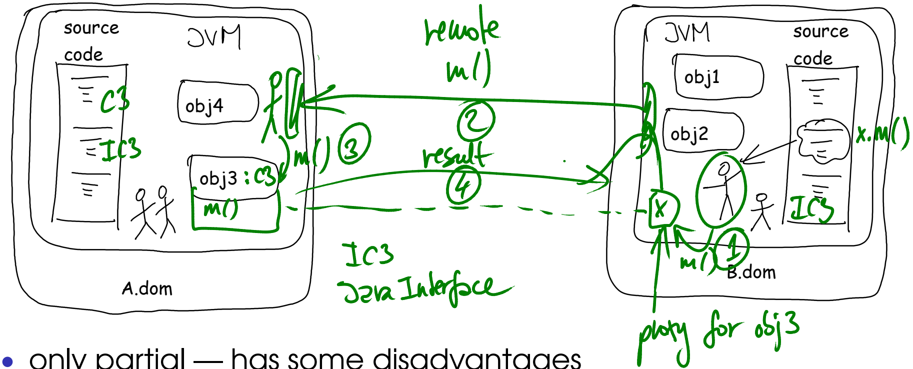
Contents

1. Distributed objects paradigm
   - Motivation
   - Location transparency in OO?
2. Accessing remote objects
   - Using remote interfaces
   - Method parameters
   - Serializable objects example
   - Fetching remote references
   - Establishing first contact
3. Defining remote objects
   - Class of remote objects
   - Remote objects accessed concurrently
4. Java RMI versus JMS
   - Loose feature and quantitative comparison

Distributed objects paradigm
Motivation

- RMI = remote method invocation
- goal: develop DSs using object-oriented paradigm
- why?
    - OOP works well for non-DSs
    - many good tools for OO design and development
    - OOP is very popular, wide-spread

Location transparency in OO?



- only partial — has some disadvantages
  - programmer should know about the overhead of RMI
- Java RMI:
  - obvious which objects are local/remote
  - same syntax for local/remote method invocation

Using remote interfaces

- only some forms of access can be remote:
  - cannot pass local references to objects
  - cannot access fields, only methods
- no need to know the full class of the object
  only its *remote interface*:
  - shared by all users of the object
  - declared in the defining class
- remote nodes should share as little code as possible
  (loose coupling)

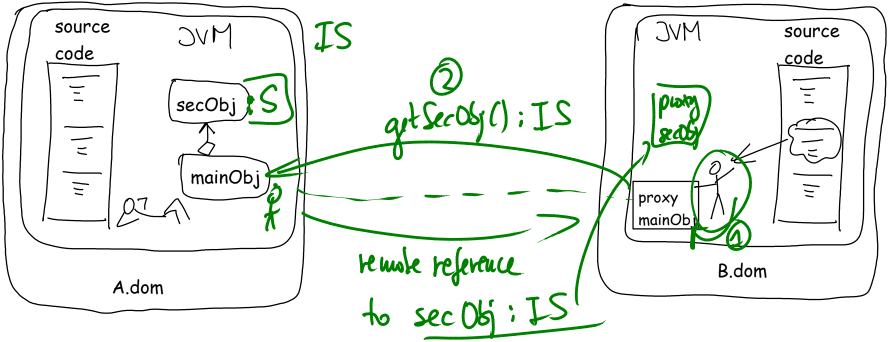
Method parameters

- parameters passed either:
  - by reference:
    - must be a remote reference
    - parameter must be a remote object (remote-enabled)
  - by value:
    - eg `int`, `char`
    - also any non-remote object
      must implement interface `Serializable`

Serializable objects example

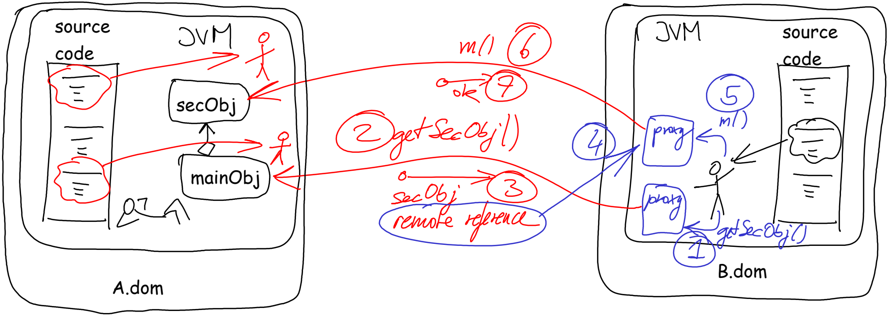
```
import java.io.Serializable;

public class Message implements Serializable
{
    private static final long serialVersionUID =
        220112709756253576L;
    private String sender;
    private String content;

    public Message(String sender, String content)
    {
        this.sender = sender;
        this.content = content;
    }

    public String toString()
    {
        return "From " + sender + ": " + content;
    }
}
```

Fetching remote references



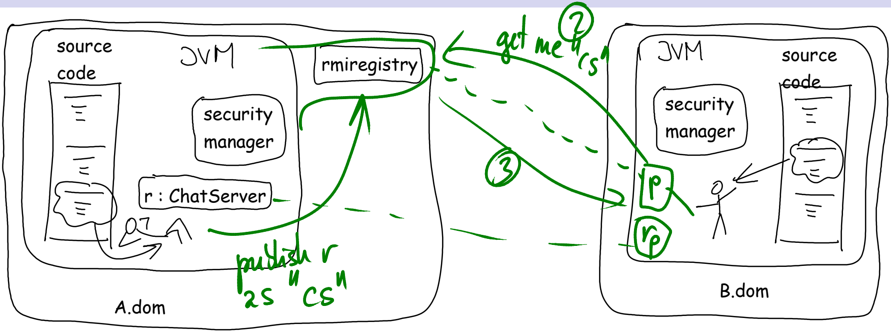
- remote objects can fetch or pass references to other remote objects

Fetching remote references



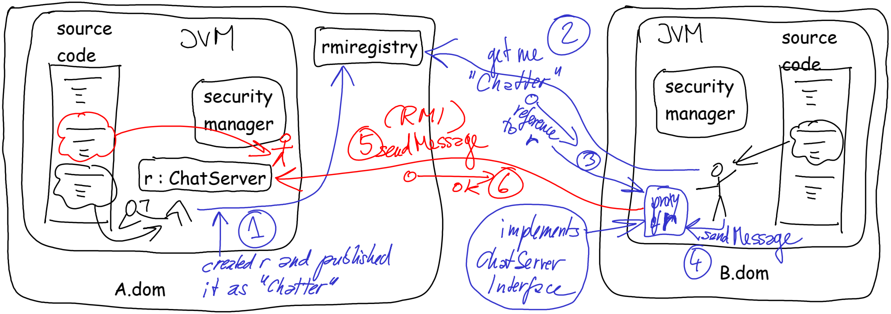
- remote objects can fetch or pass references to other remote objects

Establishing first contact



- register remote object *r* with rmiregistry under a name
- obtain a remote reference to *r* via rmiregistry
- a proxy for *r* gets created locally
- remote method invocation via proxy

Establishing first contact



- register remote object *r* with rmiregistry under a name
- obtain a remote reference to *r* via rmiregistry
- a proxy for *r* gets created locally
- remote method invocation via proxy

Defining remote objects

Class of remote objects

- class must extend `java.rmi.UnicastRemoteObject`
- is automatically `Serializable`
- must implement at least one `Remote` interface
- each constructor must call parent's constructor (using **`super`**):

```
public class ChatServer
    extends UnicastRemoteObject
    implements ChatServerInterface
{
    private static final long serialVersionUID =
        -1140073548213973798L;

    public ChatServer() throws RemoteException
    {
        super();
    }
...
```

Remote objects accessed concurrently

- each remote access — possibly different thread
- need to synchronise:
  - each remote access to an object's field (both read and write)
  - unless the field is constant (read-only)
- try not to block for long
  - not always synchronise all remote methods:

```
public void subscribe(ChatClientInterface client)
    throws RemoteException
{
    synchronized(listeners) { listeners.add(client); }
    System.out.printf("subscribed client: %s\n", client.getName());
}
```

Java RMI versus JMS
Loose feature and quantitative comparison

aspect	***Java RMI***	***JMS***
message timing	synchronous	asynchronous
remote interface	explicit as shared Java interface	implicit — programmer must check that sent messages can be received
neighbour discovery	needs registry	automatic on LAN
ease of synchronisation	difficult to get right	a little bit easier

Learning Outcomes

Learning Outcomes. You should now be able to

- read and modify existing Java RMI applications
- write simple Java RMI applications correctly, in particular:
  - program initial contact to a remote object
  - exchange remote references to objects
  - exchange serialisable parameters
  - synchronise remote access to the state of remote objects
- discuss the differences between JMS and Java RMI