Module 23: Spring RMI

CS544: Enterprise Architecture

Spring RMI

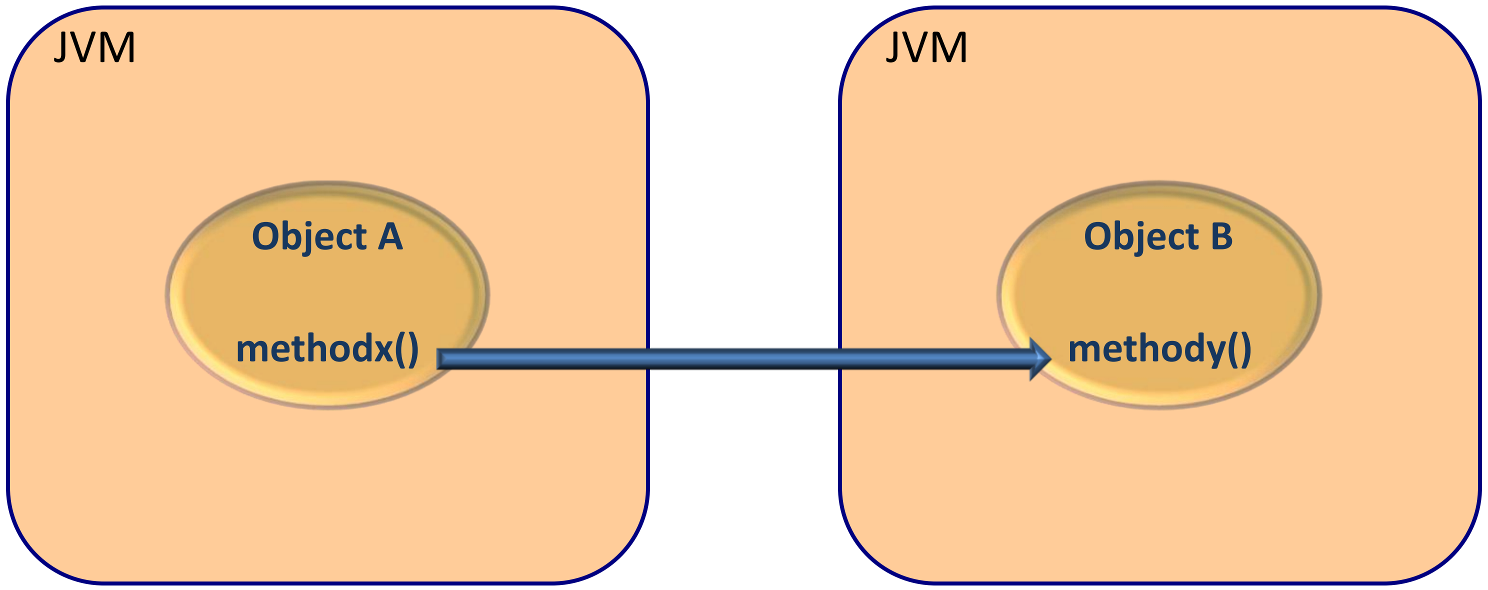
- In this module we will first take a look at what RMI is, and define some of the terms associated with RMI.
- After which we will look at an normal RMI application, and then one with the Spring template, demonstrating how the template simplifies RMI.
- Lastly we will finish up with a discussion on Spring RMI and concurrency

Spring RMI:

SPRING RMI

RMI

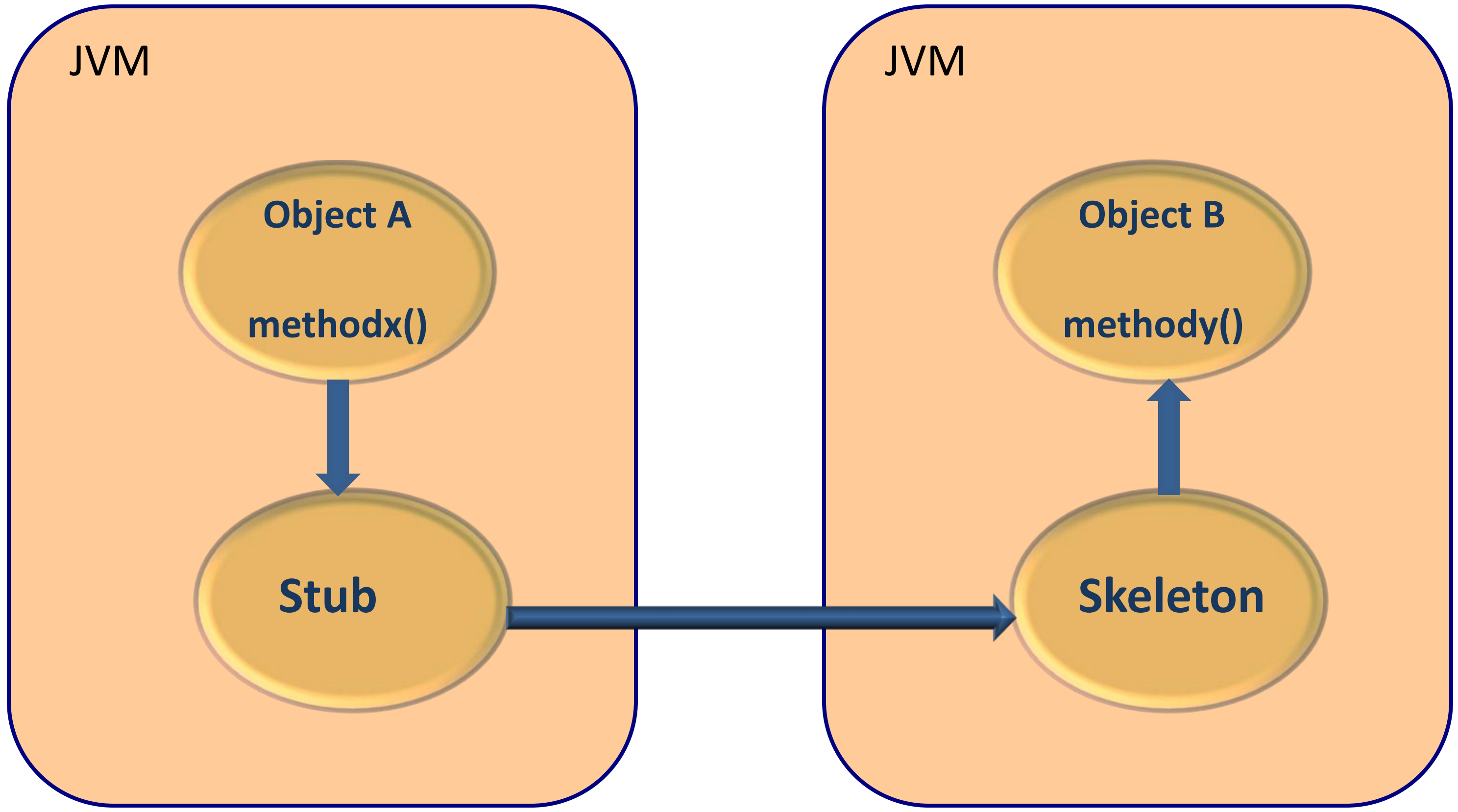
- An object calls a method of another object that lives in a different virtual machine.



Characteristics of RMI

- Synchronous
  - The calling object has to wait until the remote method call returns
- Call by value
  - If the remote method needs other objects as parameters, these parameter objects will be serialized and will be sent to the remote object.
  - All associated object will also be serialized.

Stub and skeleton



RMI Server

```java
public interface HelloServer extends Remote {
  public String sayHello(Person person) throws RemoteException;
}
```

Remote methods can throw a RemoteExeption

```java
public class HelloServerImpl extends UnicastRemoteObject implements HelloServer{
  public HelloServerImpl() throws RemoteException {
    super();
  }
  public String sayHello(Person person) {
    System.out.println("Calling HelloServerImpl with "+person.getFirstname()+" "+
                            person.getLastname());
    return "Hello "+person.getFirstname()+" "+ person.getLastname();
  }
}
```

Remote objects need to extend UnicastRemoteObject

```java
public class Person implements Serializable{
    private String firstname;
    private String lastname;

    public Person(String firstname, String lastname) {
        this.firstname=firstname;
        this.lastname=lastname;
    }
    ...
}
```

Objects that are send over the wire need to be serializable

RMI Server application

```
public class RmiServerApplication {
  public static void main(String[] args) throws Exception{
    int registryPortNumber = 1099;
    if (System.getSecurityManager() == null) {
      System.setSecurityManager(new RMISecurityManager());
    }
    // Start RMI registry
    LocateRegistry.createRegistry(registryPortNumber);
    // create server object and bind it in the registry
    HelloServer serverObject = new HelloServerImpl();
    Naming.rebind("HelloServer", serverObject);
    System.out.println("The HelloServer is running...");
  }
}
```

Use a RMISecurityManager

Start the RMI registry

Bind the server in the registry

RMI Client application

```java
public class RmiClientApplication {
  public static void main(String[] args) throws Exception{
    String host = "localhost";
    int portNumber = 1099;
    String lookupName = "//" + host + ":" + portNumber + "/" + "HelloServer";
    HelloServer remoteServerObject = (HelloServer) Naming.lookup(lookupName);
    String result = remoteServerObject.sayHello(new Person("Frank", "Brown"));
    System.out.println("RMI Client: " + result);
  }
}
```

Spring RMI Server

```java
public interface HelloServer  {
   public String sayHello(Person person);
}
```

The interface is a Plain Old Java Interface (POJI)

```java
public class HelloServerImpl implements HelloServer{

  public String sayHello(Person person) {
    System.out.println("Calling HelloServerImpl with "+person.getFirstname()+" "+
                        person.getLastname());
    return "Hello "+person.getFirstname()+" "+ person.getLastname();
  }
}
```

Remote objects is a POJO

```java
public class Person implements Serializable{
    private String firstname;
    private String lastname;

    public Person(String firstname, String lastname) {
        this.firstname=firstname;
        this.lastname=lastname;
    }
    ...
}
```

Objects that are send over the wire need to be serializable

Spring RMI Server application

```java
public class RmiServerApplication {
  public static void main(String[] args) throws Exception{
    ApplicationContext context = new ClassPathXmlApplicationContext("springconfigserver.xml");
    System.out.println("The HelloServer is running...");
  }
}
```

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans …

  <bean id="rmiServer" class="org.springframework.remoting.rmi.RmiServiceExporter">
    <property name="service" ref="helloServer" />
    <property name="serviceName" value="helloServer" />
    <property name="registryPort" value="1099" />
    <property name="serviceInterface" value="rmiserver.HelloServer" />
  </bean>
  <bean id="helloServer" class="rmiserver.HelloServerImpl"></bean>
</beans>
```

Spring RMI Client application

```java
public class RmiClientApplication {
  public static void main(String[] args) throws Exception{
    ApplicationContext context = new ClassPathXmlApplicationContext("springconfigclient.xml");
    HelloServer remoteServerObject = context.getBean("helloserver",HelloServer.class);
    String result = remoteServerObject.sayHello(new Person("Frank", "Brown"));
    System.out.println("RMI Client: " + result);
  }
}
```

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans …

  <bean id="helloserver" class="org.springframework.remoting.rmi.RmiProxyFactoryBean">
    <property name="serviceUrl" value="rmi://localhost:1099/helloServer" />
    <property name="serviceInterface" value="rmiserver.HelloServer" />
  </bean>
</beans>
```

Spring RMI

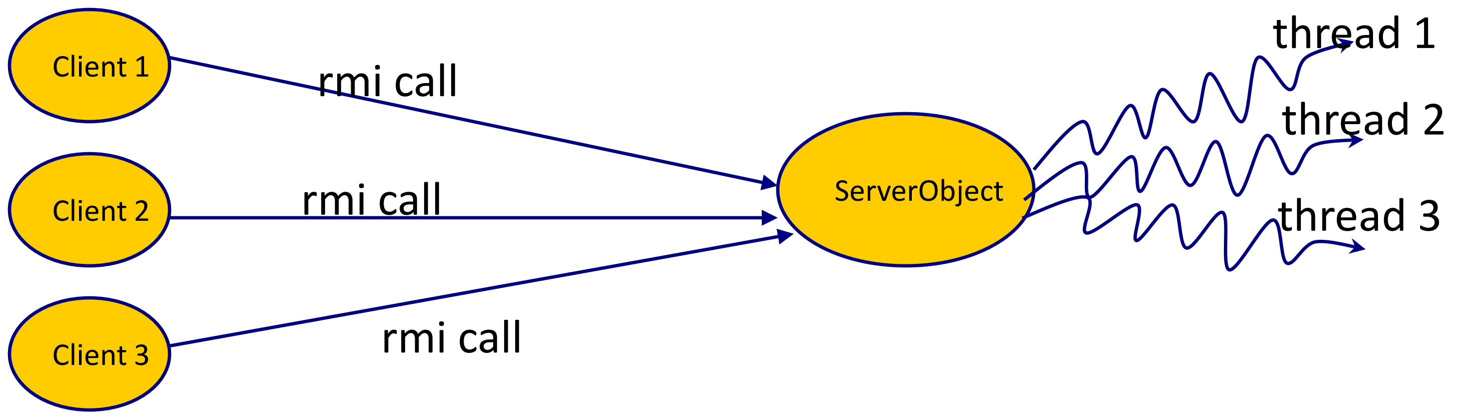
- Spring simplifies RMI development by no longer requiring your classes to be tied to RMI specific classes – Any POJO can be an RMI server.
- Spring makes the creation of RMI clients and servers a simple matter of spring configuration
  - No RMI specific code in the server classes
  - No RMI specific code in the client classes
  - To the client the server is just another bean, the fact that RMI occurs is completely hidden.

Spring RMI:

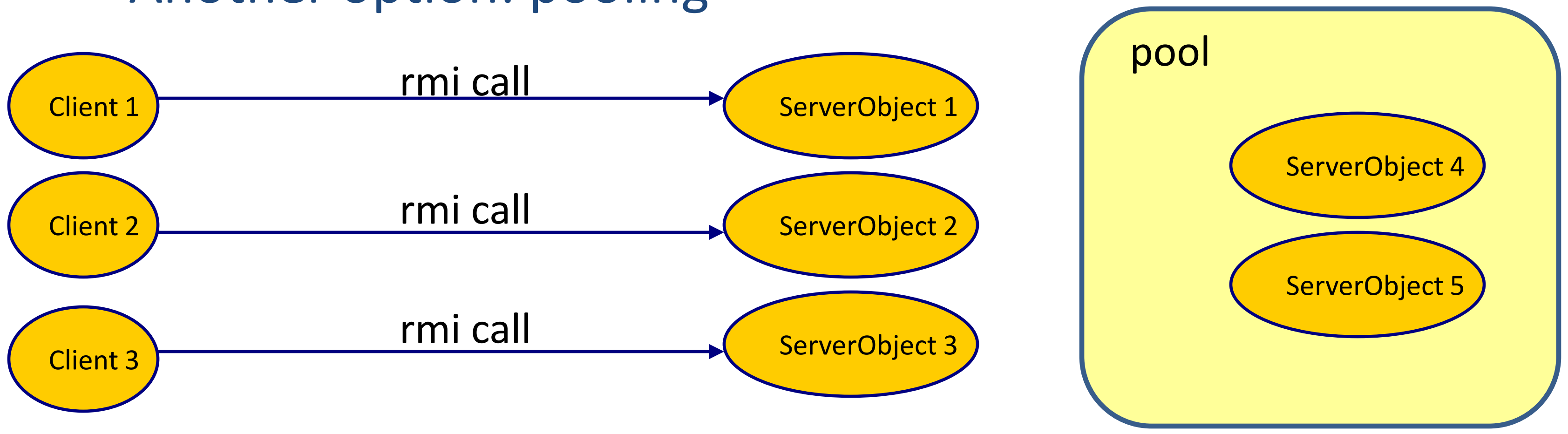
RMI AND CONCURRENCY

RMI and concurrency

- Every remote method call executes in its own thread



- Another option: pooling



Thread safety

- A method is not thread-safe if it writes to instance variables (or calls other non thread-safe methods).
- Example:

```
public class Calculator {
  private int currentValue=0;

  public int add (int value){
    currentValue=currentValue+value;
    return currentValue;
  }
  public int subtract (int value){
    currentValue=currentValue-value;
    return currentValue;
  }
}
```

The instance variable currentValue is changed

The instance variable currentValue is changed

Make code thread-safe

- There are 2 options to make code thread save:
  1. Use the ‘synchronized’ keyword
  2. Do not use instance variables

1. The ‘synchronized’ keyword

- Serial execution of synchronized methods
- Disadvantage:
  - Performance can be an issue
  - Deadlock

```
public class Calculator {
  private int currentValue=0;

  public synchronized int add (int value){
    currentValue=currentValue+value;
    return currentValue;
  }
  public synchronized int subtract (int value){
    currentValue=currentValue-value;
    return currentValue;
  }
}
```

Synchronized method

Synchronized method

2. Do not use instance variables

- Make the service stateless
- If you need state
  - Store the state in the database
  - Synchronize database access with transactions

```
public class Calculator {
  private AccountDAO accountDao;

  public void setAccountDao(AccountDAO accountDao) {
    this.accountDao = accountDao;
  }
  public int add (int value){
    int currentValue = accountDao.loadValue();
    currentValue=currentValue+value;
    accountDao.updateValue(currentValue);
    return currentValue;
  }
  public int subtract (int value){
    int currentValue = accountDao.loadValue();
    currentValue=currentValue-value;
    accountDao.updateValue(currentValue);
    return currentValue;
  }
}
```

Get the calculator value from the database

RMI and pooling

```
<?xml version="1.0" encoding="UTF-8"?>
<beans …

  <bean id="rmiServer" class="org.springframework.remoting.rmi.RmiServiceExporter">
    <property name="service" ref="helloServer" />
    <property name="serviceName" value="helloServer" />
    <property name="registryPort" value="1099" />
    <property name="serviceInterface" value="rmiserver.HelloServer" />
  </bean>
  <bean id="helloServerTarget" class="rmiserver.HelloServerImpl" scope="prototype"/>

  <bean id="poolTargetSource" class="org.springframework.aop.target.CommonsPoolTargetSource">
    <property name="targetBeanName" value="helloServerTarget"/>
    <property name="maxSize" value="25"/>
  </bean>

   <bean id="helloServer" class="org.springframework.aop.framework.ProxyFactoryBean">
     <property name="targetSource" ref="poolTargetSource"/>
   </bean>
</beans>
```

Add scope="prototype"

A pool of maximal 25 objects

helloServer is a ProxyFactoryBean and the targetSource is a pool of 25 HelloServerImpl objects

RMI and Concurrency

- Since every call will run in its own thread your application could face race conditions unless:
  - You make your code thread safe:
    - No instance variables
    - Or use synchronized
  - Use pooling to provide each calling thread with its own copy of the RMI server
    - With Spring it's easy to configure a pool for any bean

Active Learning

- Why is it important to make methods that can be called over RMI thread safe?

- Arguments that are passed to an RMI method have implement Serializable, why is that?

Summary

- Spring makes it very easy to make any method of any POJO a remote method
  - You only have to configure it in the XML file
- Spring makes it very easy to call a remote method
  - You only have to configure it in the XML file
- Make sure that remote methods are thread-safe
  - Sychronized keyword
  - Stateless remote object
- Spring makes it easy to pool any POJO
  - You only have to configure it in the XML file