Praktische Softwaretechnologie

Károly Bósa
(Karoly.Bosa@jku.at)

Research Institute for Symbolic Computation
(RISC)

A Short Introduction to Concurrency

Karoly.Bosa@jku.at

In concurrent programming, there are two basic units of execution: *processes* and *threads*.

- A **process** has a self-contained execution environment. A process generally has a complete, private set of basic run-time resources; in particular, each process has its own memory space.

- **Threads** are sometimes called *lightweight processes*. Threads share the process's resources, including memory and open files. This makes efficient, but potentially problematic, communication. Every application has at least one thread, called the *main thread*

A Short Introduction to Thread

An application that creates a thread must provide the code that will run in that thread (concurrently with the main thread). There are two ways to do this:

- Runnable interface (method *run()*)
- Subclass of java.lang.Thread

If an applet performs a time-consuming task, it should create and use its own thread to perform that task.

Running a Thread

Karoly.Bosa@jku.at

- Define class that implements Runnable interface (run method)
- Place code for the task into the run method
- Create an object of the class
- Construct a Thread Object and supply the Runnable object in the constructor
- Call the start method of the thread object

An Example for the Interface Runnable

Karoly.Bosa@jku.at

```
public class HelloRunnable implements Runnable
{
        public void run() {
                System.out.println("Hello from a thread!")
                try {
                        Thread.sleep(4000);  //Sleeping for 4000 milliseconds
                } catch(InterruptedException e){}
                System.out.println("The thread finishes its activity");
        }

        public static void main(String args[]) {
                (new Thread(new HelloRunnable())).start();
                 System.out.println("Hello from the main thread!");
                //…other activities of the main thread
                System.out.println("The main thread finishes its activity");
        }
}
```

Running two Threads

Karoly.Bosa@jku.at

```
public class Greeting implements Runnable {

private String name;

public Greeting(String n) {
  name = n;
}

public void run( ) {
  try {
    for (int i = 0; i < 12; i++) {
      System.out.println(i + " : Hello " + name);
      Thread.sleep(2000);
    }
  }
  catch (InterruptedException e) {}
}
}
```

```
public class TestGreeting {

public static void main(String[] args) {
  Runnable g1 = new Greeting("Tom");
  Runnable g2 = new Greeting("Lisa");
  Thread tg1 = new Thread(g1);
  Thread tg2 = new Thread(g2);
  tg1.start();
  tg2.start();
}
}
```

Thread Subclass

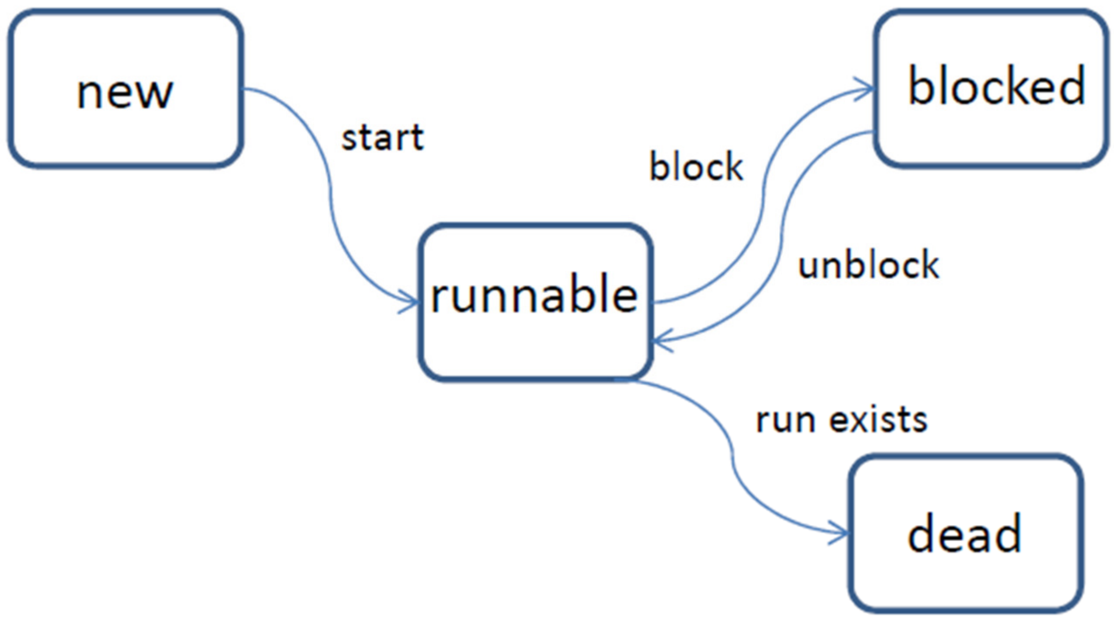
Karoly.Bosa@jku.at

```
public class GreetingThread extends Thread {
private String name;

public GreetingThread(String n) {
  name = n;
}

public void run() {
  try {
    for (int i = 0; i < 12; i++) {
      System.out.println(i + " : Hello " + name);
      Thread.sleep(2000);
    }
  }
  catch (InterruptedException e) {}
}
}
```

```
public class TestGreetingThread {

public static void main(String[] args) {
  Thread t1 = new GreetingThread("Tom");
  Thread t2 = new GreetingThread("Lisa");
  t1.start();
  t2.start();
}
}
```

Thread States

Karoly.Bosa@jku.at



Blocked State

- Sleeping
- Waiting for input/output (IO-blocked)
- Waiting to acquire a lock (locked)
- Waiting for a condition (waiting)

Terminating Threads

Karoly.Bosa@jku.at

- Thread terminates when the run method of its Runnable returns.
- Notify the threads that they should terminate itself -> interrupt() method

```
public class DemoInterrupt implements Runnable {
  public void run() {
    while(!Thread.interrupted()) {
      System.out.println("Endless loop!");
    }
  }
}
```

Pausing Execution

Karoly.Bosa@jku.at

- Thread.sleep causes the current thread to suspend execution for a specified period.
- Makes processor time available to other threads

```
try {
  Thread.sleep(DELAY);
}
catch (InterruptedException e) {
  Thread.currentThread().interrupt();
}
```

Thread Synchronization

Karoly.Bosa@jku.at

- When threads share access to a common object, they can conflict with each other!
- Locks
- Synchronized methods

Synchronized Methods

Karoly.Bosa@jku.at

```
public class SynchronizedExample
{
        private int a = 5;
        private int b = 12;
        public void exchangeValues()
        {
                a = a+b;
                b = a-b;        //Let b equal the previous value of a
                a = a-b;        //Let a equal the previous value of b
        }
        public void decrement()
        {
                a--; b--;
        }

        public int valueOfA()
        {return a; }
}
```

Synchronized Methods (Atomic operation)

Karoly.Bosa@jku.at

```
public class SynchronizedExample
{
        private int a = 5;
        private int b = 12;
        public synchronized void exchangeValues()
        {
                a = a+b;
                b = a-b;            //Let b equal the previous value of a
                a = a-b;            //Let a equal the previous value of b
        }
        public synchronized void decrement()
        {
                a--; b--;
        }

        public synchronized int valueOfA()
        {return a; }
}
```

Race Condition

Karoly.Bosa@jku.at

- A race condition occurs if the effect of multiple threads on shared data depends on the order in which the threads are sheduled.
- Solution : Locks
  - ReentrantLock
  - Locks that are built into every Java Object

ReentrantLock

Karoly.Bosa@jku.at

- Block of code is exclusively executed by a single thread

```
aLock = new ReentrantLock();
...
aLock.lock();
try {
// protected code
}
finally {
aLock.unlock();
}
```

Object Lock

Karoly.Bosa@jku.at

- Every Java Object has an associated object lock.
- Synchronized method
- Synchronized block

Synchronized Method

Karoly.Bosa@jku.at

- To make a method synchronized, simply add the synchronized keyword to its declaration
- When one thread is executing a synchronized method for an object, all other threads that invoke synchronized methods for the same object block until the first thread is done with the object.

```
public synchronized void  transfer(int from, int to, int amount) {
    int tempAmount;

    tempAmount = account[from];
    tempAmount -= amount;
    ....

}
```

Syncronized Statements

Karoly.Bosa@jku.at

```
public void addName(String name)
{
        synchronized(this)
        {
                lastName = name;
                nameCount++;
        }
        nameList.add(name);
}
```

- Synchronized statements must specify the object that provides the *lock*
- Consequence: from the synchronized block, you must not call other objects' methods (out of the scope of the lock).

Wait and Notify

Karoly.Bosa@jku.at

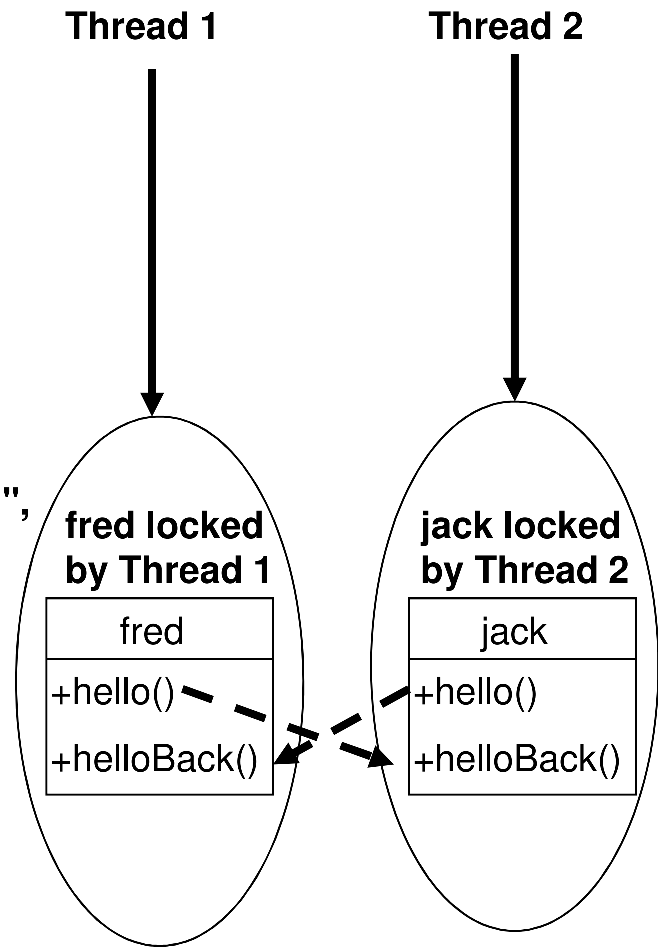
- Guarded Blocks: Such a block begins by polling a condition that must be true before the block can proceed
- wait: suspend the current thread. wait does not return until another thread has issued a notification

```
class Bank {
  static int[] account = {30, 50, 100};

  public void  transfer(int from, int to, int amount) throws InterruptedException {
   int tempAmount;

   while(account[from] < amount)
     wait();

   tempAmount = account[from];
   tempAmount -= amount;
   ....
  account[to] = tempAmount;
   notifyAll();
  }
 ...
}
```

Deadlock

Karoly.Bosa@jku.at

```
public class Deadlock {
  static class Friend {
   private final String name;
   public Friend(String name) { this.name = name; }
   public String getName() { return this.name; }
   public synchronized void hello(Friend friend) {
              System.out.format("%s: %s has said hello to me!%n",
                          this.name, friend.getName());
              friend.helloBack(this);
   }

   public synchronized void helloBack(Friend friend) {
              System.out.format("%s: %s has said hello back to me!%n",
                          this.name, friend.getName());
   }
  }

  public static void main(String[] args) {
   final Friend fred = new Friend("Jack");
   final Friend jack = new Friend("Fred");
   new Thread() { public void run() { fred.hello(jack); } }.start();
   new Thread() { public void run() { jack.hello(fred); } }.start();
 }
}
```



Deadlock **describes a situation where two or more threads are blocked forever, waiting for each other.**

Join

Karoly.Bosa@jku.at

- join method allows one thread to wait for the completion of another.
- If t is a Thread object whose thread is currently executing t.join() causes the current thread to pause execution until t's thread terminates

```
class JoinTheThread {
static class JoinerThread extends Thread   {
  public int result;
  public void run() {
    result = 1;
  }
}
public static void main( String[] args ) throws InterruptedException {
  JoinerThread t = new JoinerThread();
  t.start();
  //t.join();
  System.out.println( t.result );
}
}
```

Socket Server with Threads

Karoly.Bosa@jku.at

```
import java.io.*;
import java.net.ServerSocket;
import java.net.Socket;

public class ServerWithThreads {
  private static class ClientHandler extends Thread{
      Socket s;
      public ClientHandler(Socket s){
         this.s=s;
      }

      public void run() {
         try {
            System.out.println("New client connected");
            BufferedReader br = new BufferedReader(
               new InputStreamReader(s.getInputStream()));

            String string = br.readLine();
            System.out.println(string);
            PrintStream pw = new
               PrintStream(s.getOutputStream());
            pw.println(string);
            s.close();
         } catch (IOException e) {
            e.printStackTrace();
         }
      }
}

public static void main(String[] args) {
      System.out.println("The server is started.");
      try {
        ServerSocket ss = new ServerSocket(3131);
        while (true) {
           Socket s = ss.accept();
           ClientHandler ch = new ClientHandler(s);
           ch.start();
        }

      } catch (IOException e) {
          e.printStackTrace();
      }
   }
}
```

Socket Client

Karoly.Bosa@jku.at

```
import java.io.*;
import java.net.Socket;
import java.net.UnknownHostException;

public class Client {
   public static void main(String[] args) {
     try {
              Socket s =new Socket("127.0.0.1",3131);
              PrintStream pw = new PrintStream(s.getOutputStream(),false);
              pw.println("Hello I am a client.");
              System.out.println("A message sent\nEcho: ");
              BufferedReader br=new BufferedReader(new InputStreamReader(s.getInputStream()));
              System.out.println(br.readLine());
     } catch (UnknownHostException e) {
              e.printStackTrace();
     } catch (IOException e) {
              e.printStackTrace();
     }
   }
}
```

Nothing changed!

Java Remote Method Invocation (RMI)

Karoly.Bosa@jku.at

- Java RMI allows an object running in one (Java virtual) machine to invoke methods of an object running in another (Java virtual) machine.

- By RMI programs written in the Java programming language are able to communicate each other.

- RMI applications (are **distributed object applications** which) often comprise two separate programs, a **server** and a **client**.

- A typical server program creates some **remote objects**, makes references to these objects accessible, and waits for clients to invoke methods on these objects.

RMI Requirements

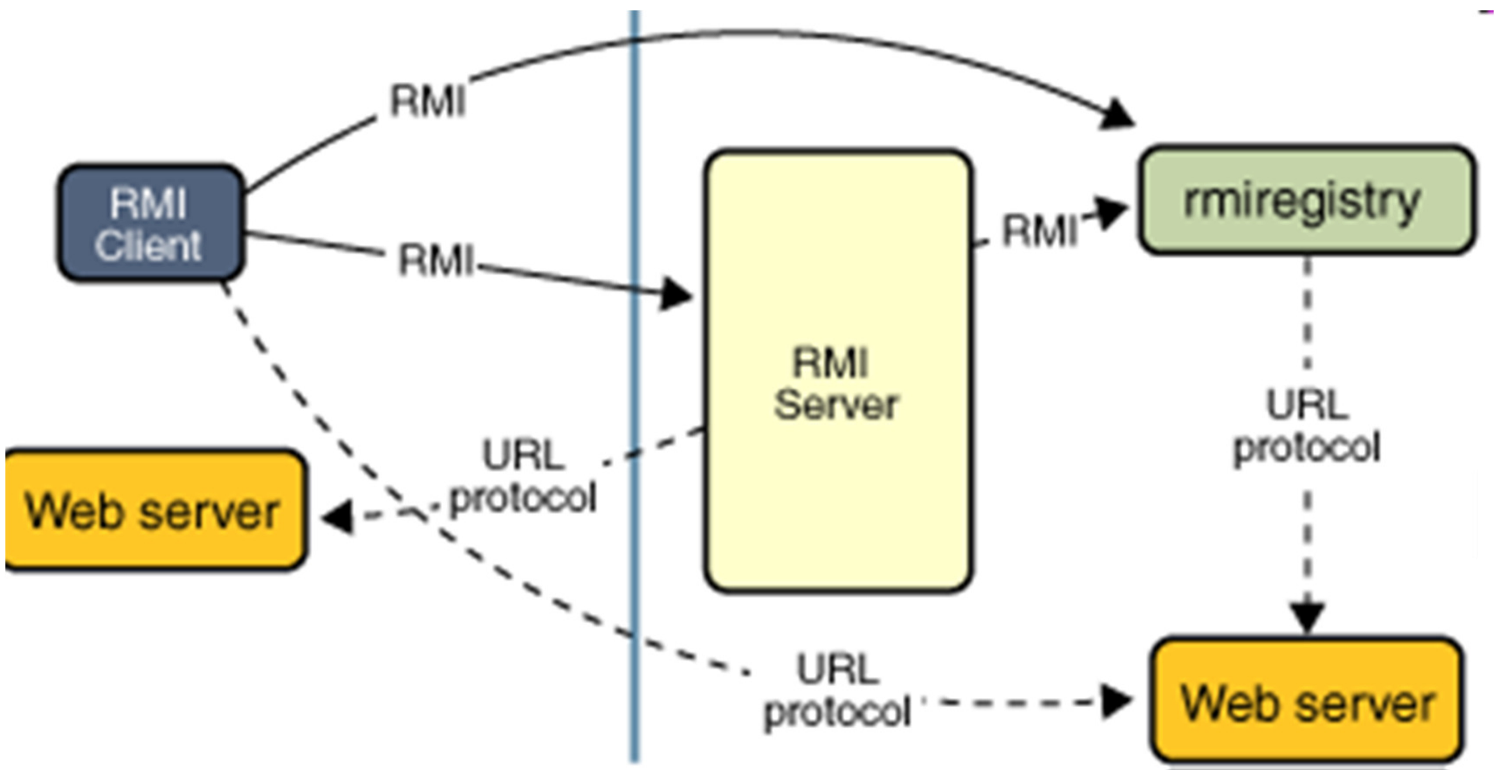
Locate remote objects. Applications can use various mechanisms to obtain references to remote objects (e.g.: query the **RMI registry**).

Communicate with remote objects. Details of communication between remote objects are handled by RMI.

Load class definitions for objects that are passed around. Because RMI enables objects to be passed back and forth, it provides mechanisms for loading an object's class definitions as well as for transmitting an object's data.

Overview of RMI

Karoly.Bosa@jku.at



Dynamic Code Loading: All of the types and behavior of an object available only in a single Java virtual machine, can be transmitted to another, possibly remote, Java virtual machine.

Remote Objects and Interfaces

- Those objects that can be invoked across Java virtual machines are called ***remote objects***.
- Remote object must always implement a **remote interface**:
  - A remote interface extends the interface **java.rmi.Remote.**
  - Each method of the interface declares **java.rmi.RemoteException** in its throws clause, in addition to any application-specific exceptions.

RMI Stubs

Karoly.Bosa@jku.at

- Rather than making a copy of the implementation object in the receiving Java virtual machine, RMI passes a remote ***stub*** for a remote object (it acts as the local representative, or proxy).

- The stub is responsible for carrying out the method invocation on the remote object.

- A stub implements the same set of remote interfaces that the remote object implements.

- ***only* those methods which are defined in a remote interface are available to be called from the remote Java virtual machine.**

Design and Implementation

- **Defining the remote interfaces.** A remote interface specifies the methods that can be invoked remotely by a client.

- **Implementing the remote objects.** Remote objects must implement one or more remote interfaces.

- **Implementing the client and the server.**

Remarks:

- As with any Java program, you use the **javac** compiler to compile the source files.
- Starting the application includes running the **RMI remote object registry**, the **server**, and the **client**.

Simple Example: Hello World

This is a simple RMI system with a client and a server.

The server contains one method (*helloWorld*) that returns a string to the client.

Each class implementation is available on both sides.

RMI Interface: HelloWorld.java


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

public interface HelloWorld extends Remote {
  String helloWorld() throws RemoteException;
}
```

RMI Server: HelloWorldServer.java

Karoly.Bosa@jku.at

```
import java.rmi.Naming;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

public class HelloWorldServer extends UnicastRemoteObject implements
      HelloWorld {
  public HelloWorldServer() throws RemoteException {
    super();
  }

  public String helloWorld() {
    System.out.println("Invocation to helloWorld was succesful!");
    return "Hello World from RMI server!";
  }

  public static void main(String args[]) {
     …
  }
}
```

RMI Server: HelloWorldServer.java

Karoly.Bosa@jku.at

```
public static void main(String args[]) {
  try {
    HelloWorldServer obj = new HelloWorldServer();
    Naming.rebind("HelloWorld", obj);
    System.out.println("HelloWorld bound in registry");
  }
  catch (Exception e) {
    System.out.println("HelloWorldServer error: " + e.getMessage());
    e.printStackTrace();
  }
}
```

It is not necessary to have a thread wait or loop to keep the server alive. As long as there is a reference to the server object in another Java virtual machine local or remote (e.g.: rmiregistry), the object will not be shut down or garbage collected.

RMI Client: HelloWorldClient.java

Karoly.Bosa@jku.at

```
import java.rmi.Naming;
import java.rmi.RemoteException;

public class HelloWorldClient {
  static String message = "blank";
  static HelloWorld obj = null;
  public static void main(String args[])
  {
    try {
      obj = (HelloWorld)Naming.lookup("//"+ "127.0.0.1"+ "/HelloWorld");
      message = obj.helloWorld();
      System.out.println("Message from the RMI-server was: \""+ message +
     "\"");
    }
    catch (Exception e) {
      System.out.println("HelloWorldClient exception: "+ e.getMessage());
      e.printStackTrace();
    }
  }
}
```

RMI Example: Execution

Karoly.Bosa@jku.at

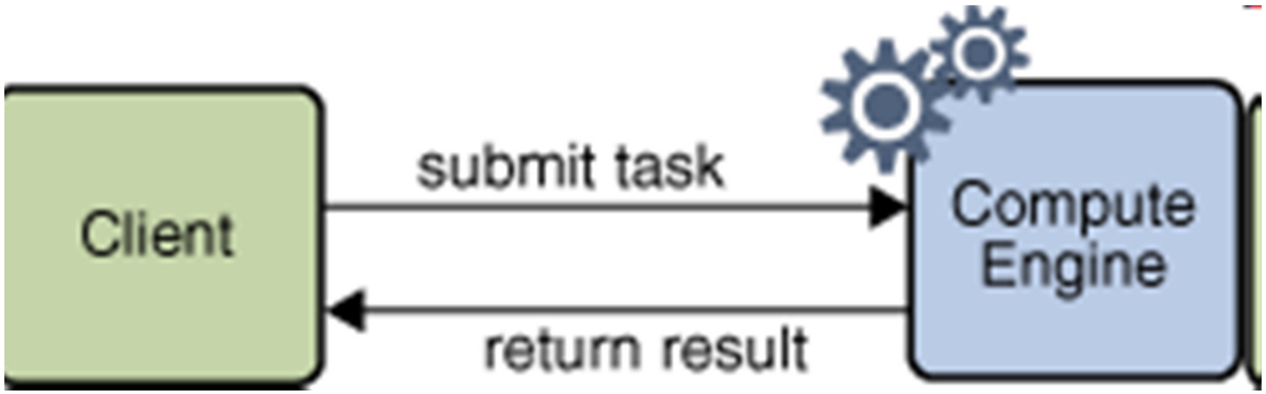
- rmiregistry (start rmiregistry)
- java HelloWorldServer
- java HelloWorldClient

For security reasons, an application can only bind, unbind, or rebind remote object references with a registry running on the same host.

RMI Example 2: Compute Engine

Karoly.Bosa@jku.at

An example from the official Java tutorial:



A simple distributed application framework:

- There is a server machine with relatively big performance factor.
- The *compute engine* of the server is a remote object on the server that is able to take **arbitrary(!)** tasks from clients, runs the tasks, and returns any results.
- **Novel aspect: the class definition of the tasks is not needed to define for the compute engine!!!** The only requirement of a task is that its class implement a particular interface.
- **The code** of the particular tasks **can be downloaded** by the RMI system **to the compute engine.**

RMI Example 2: Remote Interface


```
package compute;

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

public interface Compute extends Remote {
    <T> T executeTask(Task<T> t) throws RemoteException;
}
```

RMI Example 2: Interface for the Tasks

Karoly.Bosa@jku.at

```
package compute;

public interface Task<T> {
    T execute();
}
```

- The object which implement this interface is intended to transfer to the Server by its value.

- Any kinds of tasks can be accepted by a Compute object as long as they are implementations of the Task.

- The classes that implement this interface can contain any data needed for the computation of the task and any other methods needed for the computation additionally.

- Task object that were previously unknown to the compute engine are downloaded by RMI into the compute engine's Java virtual machine.

RMI Example 2: Server

Karoly.Bosa@jku.at

```
import compute.Compute;
import compute.Task;

import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.rmi.server.UnicastRemoteObject;

public class ComputeEngine implements
Compute {

    public ComputeEngine() {
        super();
    }

    public <T> T executeTask(Task<T> t) {
        return t.execute();
    }

public static void main(String[] args) {
        if (System.getSecurityManager() == null) {
            System.setSecurityManager(new
                                     SecurityManager());
        }
        try {
            String name = "Compute";
            Compute engine = new ComputeEngine();
            Compute stub =
                    (Compute) UnicastRemoteObject.
                    exportObject(engine, 0);
            Registry registry =
                    LocateRegistry.getRegistry();
            registry.rebind(name, stub);
            System.out.println("ComputeEngine bound");
        } catch (Exception e) {
            System.err.println("ComputeEngine
                    exception:");
            e.printStackTrace();
        }
    }
}
```

RMI Example 2: Server

Karoly.Bosa@jku.at

```
import compute.Compute;
import compute.Task;

import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.rmi.server.UnicastRemoteObject;

public class ComputeEngine implements
Compute {

   public ComputeEngine() {
      super();
   }

   public <T> T executeTask(Task<T> t) {
      return t.execute();
   }

public static void main(String[] args) {
      if (System.getSecurityManager() == null) {
         System.setSecurityManager(new
                                          SecurityManager());
      }
      try {
         String name = "Compute";
         Compute engine = new ComputeEngine();
         Compute stub =
               (Compute) UnicastRemoteObject.
               exportObject(engine, 0);
         Registry registry =
               LocateRegistry.getRegistry();
         registry.rebind(name, stub);
         System.out.println("ComputeEngine bound");
      } catch (Exception e) {
         System.err.println("ComputeEngine
               exception:");
         e.printStackTrace();
      }
   }
}
```

Arguments and return values of remote methods can be primitive data types, a remote objects, or a *serializable* objects.

RMI Example 2: Stubs and Serializable Objects

Karoly.Bosa@jku.at

- Remote objects are passed by reference (stub). This means:
  - any changes made to the state of the object by remote method invocations (by a client) are reflected in the original remote object (on the server), too and
  - any methods defined in the implementation of a class, but not defined by any remote interfaces implemented by this class are not available to clients.
- Objects that are not remote objects are passed by value (copy). This means:
  - By default, all fields are copied except fields that are marked static or transient,
  - any changes in the object's state by a client are reflected only in the clients's copy (not in the server's original instance) and
  - any changes in the object's state by the server are reflected only in the server's original instance, not in the copy of the clients.

RMI Example 2: Stubs and Serializable Objects

Karoly.Bosa@jku.at

- Remote objects are passed by reference (stub). This means:
  - any changes made to the state of the object by remote method invocations (by a client) are reflected in the original remote object (on the server), too and
  - any methods defined in the implementation of a class, but not defined by any remote interfaces implemented by this class are not available to clients.
- Objects that are not remote objects are passed by value (copy). This means:
  - By default, all fields are copied except fields that are marked static or transient,
  - any changes in the object's state by a client are reflected only in the clients's copy (not in the server's original instance) and
  - any changes in the object's state by the sserver are reflected only in the server's original instance, not in the copy od the clients.

RMI Example 2: Server

Karoly.Bosa@jku.at

```
import compute.Compute;
import compute.Task;

import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.rmi.server.UnicastRemoteObject;

public class ComputeEngine implements
Compute {

    public ComputeEngine() {
        super();
    }

    public <T> T executeTask(Task<T> t) {
        return t.execute();
    }

public static void main(String[] args) {
        if (System.getSecurityManager() == null) {
            System.setSecurityManager(new
                                    SecurityManager());
        }
        try {
            String name = "Compute";
            Compute engine = new ComputeEngine();
            Compute stub =
                    (Compute) UnicastRemoteObject.
                    exportObject(engine, 0);
            Registry registry =
                    LocateRegistry.getRegistry();
            registry.rebind(name, stub);
            System.out.println("ComputeEngine bound");
        } catch (Exception e) {
            System.err.println("ComputeEngine
                    exception:");
            e.printStackTrace();
        }
    }
}
```

If an RMI program does not install a security manager, RMI will not download classes (other than from the local class path)

RMI Example 2: Server

Karoly.Bosa@jku.at

```
import compute.Compute;
import compute.Task;

import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.rmi.server.UnicastRemoteObject;

public class ComputeEngine implements
Compute {

    public ComputeEngine() {
        super();
    }

    public <T> T executeTask(Task<T> t) {
        return t.execute();
    }

public static void main(String[] args) {
    if (System.getSecurityManager() == null) {
        System.setSecurityManager(new
                                  SecurityManager());
    }
    try {
        String name = "Compute";
        Compute engine = new ComputeEngine();
        Compute stub =
            (Compute) UnicastRemoteObject.
            exportObject(engine, 0);
        Registry registry =
            LocateRegistry.getRegistry();
        registry.rebind(name, stub);
        System.out.println("ComputeEngine bound");
    } catch (Exception e) {
        System.err.println("ComputeEngine
            exception:");
        e.printStackTrace();
    }
  }
}
```

The *exportObject* method returns a stub for the exported remote object whose type is *Compute*.

RMI Example 2: Server

Karoly.Bosa@jku.at

```
import compute.Compute;
import compute.Task;

import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.rmi.server.UnicastRemoteObject;

public class ComputeEngine implements
Compute {

    public ComputeEngine() {
        super();
    }

    public <T> T executeTask(Task<T> t) {
        return t.execute();
    }

  public static void main(String[] args) {
      if (System.getSecurityManager() == null) {
          System.setSecurityManager(new
                                  SecurityManager());
      }
      try {
          String name = "Compute";
          Compute engine = new ComputeEngine();
          Compute stub =
              (Compute) UnicastRemoteObject.
              exportObject(engine, 0);
          Registry registry =
              LocateRegistry.getRegistry();
          registry.rebind(name, stub);
          System.out.println("ComputeEngine bound");
      } catch (Exception e) {
          System.err.println("ComputeEngine
              exception:");
          e.printStackTrace();
      }
  }
}
```

RMI Example 2: Client

Karoly.Bosa@jku.at

```
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.math.BigDecimal;
import compute.Compute;

public class ComputePi {
   public static void main(String args[]) {
      if (System.getSecurityManager() == null) {
         System.setSecurityManager(new SecurityManager());
      }
      try {
         String name = "Compute";
         Registry registry = LocateRegistry.getRegistry(args[0]);
         Compute comp = (Compute) registry.lookup(name);
         Pi task = new Pi(Integer.parseInt(args[1]));
         BigDecimal pi = comp.executeTask(task);
         System.out.println(pi);
      } catch (Exception e) {
         System.err.println("ComputePi exception:");
         e.printStackTrace();
      }
   }
}
```

RMI Example 2: Client

Karoly.Bosa@jku.at

```
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.math.BigDecimal;
import compute.Compute;

public class ComputePi {
    public static void main(String args[]) {
        if (System.getSecurityManager() == null) {
            System.setSecurityManager(new SecurityManager());
        }
        try {
            String name = "Compute";
            Registry registry = LocateRegistry.getRegistry(args[0]);
            Compute comp = (Compute) registry.lookup(name);
            Pi task = new Pi(Integer.parseInt(args[1]));
            BigDecimal pi = comp.executeTask(task);
            System.out.println(pi);
        } catch (Exception e) {
            System.err.println("ComputePi exception:");
            e.printStackTrace();
        }
    }
}
```

RMI Example 2: Client

Karoly.Bosa@jku.at

```
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.math.BigDecimal;
import compute.Compute;

public class ComputePi {
   public static void main(String args[]) {
      if (System.getSecurityManager() == null) {
         System.setSecurityManager(new SecurityManager());
      }
      try {
         String name = "Compute";
         Registry registry = LocateRegistry.getRegistry(args[0]);
         Compute comp = (Compute) registry.lookup(name);
         Pi task = new Pi(Integer.parseInt(args[1]));
         BigDecimal pi = comp.executeTask(task);
         System.out.println(pi);
      } catch (Exception e) {
         System.err.println("ComputePi exception:");
         e.printStackTrace();
      }
   }
}
```

RMI Example 2: The Computational Task

Karoly.Bosa@jku.at

```
import java.io.Serializable;
import java.math.BigDecimal;
import compute.Task;

public class Pi implements Task<BigDecimal>, Serializable {
    private static final long serialVersionUID = 227L;
    private final int digits;       // Calculate pi to the specified  precision.
    public Pi(int digits) {
        this.digits = digits;
    }

    public BigDecimal execute() {
        return computePi(digits);
    }

     //The implementation of the particular task is unimportant for the example
    //It is a relatively time consuming algorithm
    public static BigDecimal computePi(int digits) { … }
}
```

RMI Example 2: The Coputational Task

Karoly.Bosa@jku.at

```
import java.io.Serializable;
import java.math.BigDecimal;
import compute.Task;

public class Pi implements Task<BigDecimal>, Serializable {
   private static final long serialVersionUID = 227L;
   private final int digits;       // Calculate pi to the specified  precision.
   public Pi(int digits) {
      this.digits = digits;
   }

   public BigDecimal execute() {
      return computePi(digits);
   }

    //The implementation of the particular task is irrelevant for the example
   //It is a relatively time consuming algorithm
   public static BigDecimal computePi(int digits) { … }
}
```

RMI uses the Java object serialization mechanism to transport objects by value between Java virtual machines. For this, classes of these objects must implement the **java.io.Serializable** interface.

RMI Example 2: The Policy Files

Karoly.Bosa@jku.at

The file **server.policy**:

```
grant codeBase "file:/c:/Class/tmp/RMI_Example2/server/" {
   permission java.security.AllPermission;
};
```

The file **client.policy**:

```
grant codeBase "file:/c:/Class/tmp/RMI_Example2/client/" {
   permission java.security.AllPermission;
};
```

RMI Example 2: Directory Hierarchy

Karoly.Bosa@jku.at

- **compute**
  - Compute.java
  - Task.java
- **server**
  - ComputeEngine.java
  - server.policy
- **client**
  - ComputePi.java
  - Pi.java
  - client.policy

RMI Example 2: Compilation

Karoly.Bosa@jku.at

- javac compute/*.java
  jar cvf compute.jar compute/*.class
  copy compute.jar server/
  copy compute.jar client/

- cd server
  javac -cp ./compute.jar *.java

- cd ../client
  javac -cp ./compute.jar *.java

RMI Example 2: Execution

Karoly.Bosa@jku.at

- rmiregistry
- cd server
  java -cp ./compute.jar;.
  -Djava.rmi.server.hostname= 127.0.0.1
  -Djava.security.policy=server.policy
  ComputeEngine
- cd client
  java -cp ./compute.jar;.
  -Djava.rmi.server.codebase=
  file:/c:/Class/tmp/RMI_Example2/client/
  -Djava.security.policy=client.policy
  ComputePi 127.0.0.1 45

The server host name information is used by clients when they attempt to communicate via remote method invocations.

Output: 3.14159265358979323846264338327950288419716939

RMI Example 2: Execution

Karoly.Bosa@jku.at

- rmiregistry

- cd server
  java -cp ./compute.jar;.
  -Djava.rmi.server.hostname= 127.0.0.1
  -Djava.security.policy=server.policy
  ComputeEngine

- cd client
  java -cp ./compute.jar;.
  -Djava.rmi.server.codebase=
  file:/c:/Class/tmp/RMI_Example2/client/
  -Djava.security.policy=client.policy
  ComputePi 127.0.0.1 45

Output: 3.141592653589793238462643383279502884197169399

RMI Example 2: Execution

Karoly.Bosa@jku.at

- rmiregistry

- cd server
  java -cp ./compute.jar;.
  -Djava.rmi.server.hostname= 127.0.0.1
  -Djava.security.policy=server.policy
  ComputeEngine

- cd client
  java -cp ./compute.jar;.
  -Djava.rmi.server.codebase=
  file:/c:/Class/tmp/RMI_Example2/client/
  -Djava.security.policy=client.policy
  ComputePi 127.0.0.1 45

The location where the client serves its classes (the Pi class) is given by using the java.rmi.server.codebase property

Output: 3.141592653589793238462643383279502884197169399

RMI Example 2: Execution

Karoly.Bosa@jku.at

If we start the client without the argument

-Djava.rmi.server.codebase=file:/c:/Class/tmp/RMI_Example2/client/

Then the server will not be able to accomplish task Pi and it will returns a *ClassNotFoundException: Pi*

```
C:\___Class\tmp\RMI_Example2\client>java -cp ./compute.jar;. -Djava.security.pol
icy=client.policy ComputePi 127.0.0.1 45
ComputePi exception:
java.rmi.ServerException: RemoteException occurred in server thread; nested exce
ption is:
        java.rmi.UnmarshalException: error unmarshalling arguments; nested excep
tion is:
        java.lang.ClassNotFoundException: Pi
        at sun.rmi.server.UnicastServerRef.dispatch(UnicastServerRef.java:336)
        at sun.rmi.transport.Transport$1.run(Transport.java:159)
        at java.security.AccessController.doPrivileged(Native Method)
        at sun.rmi.transport.Transport.serviceCall(Transport.java:155)
        at sun.rmi.transport.tcp.TCPTransport.handleMessages(TCPTransport.java:5
35)
```

RMI Example 2: Execution

Karoly.Bosa@jku.at

- rmiregistry

- cd server
  java -cp ./compute.jar;.
  -Djava.rmi.server.hostname= 127.0.0.1
  -Djava.security.policy=server.policy
  ComputeEngine

- cd client
  java -cp ./compute.jar;.
  -Djava.rmi.server.codebase=
  file:/c:/Class/tmp/RMI_Example2/client/
  -Djava.security.policy=client.policy
  ComputePi 127.0.0.1 45

Output: 3.141592653589793238462643383279502884197169399

RMI Example 2: Execution

Karoly.Bosa@jku.at

- rmiregistry

- cd server
  java -cp ./compute.jar;.
  -Djava.rmi.server.hostname= 127.0.0.1
  -Djava.security.policy=server.policy
  ComputeEngine

- cd client
  java -cp ./compute.jar;.
  -Djava.rmi.server.codebase=
  file:/c:/Class/tmp/RMI_Example2/client/
  -Djava.security.policy=client.policy
  ComputePi 127.0.0.1 45

Output: 3.141592653589793238462643383279502884197169399

RMI Example 2: Execution

Karoly.Bosa@jku.at

- rmiregistry

- cd server
  java -cp ./compute.jar;.
  -Djava.rmi.server.hostname= 127.0.0.1
  -Djava.security.policy=server.policy
  ComputeEngine

- cd client
  java -cp ./compute.jar;.
  -Djava.rmi.server.codebase=
  file:/c:/Class/tmp/RMI_Example2/client/
  -Djava.security.policy=client.policy
  ComputePi 127.0.0.1 45

Output: 3.141592653589793238462643383279502884197169399

RMI Example 2: Execution

Karoly.Bosa@jku.at

```
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

c:\___Class\tmp\RMI_Example2>rmiregistry
```

C:\WINDOWS\system32\CMD.exe - java -cp ./compute.jar;. -Djava.r...

```
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

c:\___Class\tmp\RMI_Example2>cd server

C:\___Class\tmp\RMI_Example2\server>java -cp ./compute.jar;. -Djava.rmi.server.h
ostname=127.0.0.1 -Djava.security.policy=server.policy ComputeEngine
ComputeEngine bound
```

C:\WINDOWS\system32\cmd.exe

```
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

c:\___Class\tmp\RMI_Example2>cd client

C:\___Class\tmp\RMI_Example2\client>java -cp ./compute.jar;. -Djava.rmi.server.c
odebase=file:/c:/___Class/tmp/RMI_Example2/client/ -Djava.security.policy=client
.policy ComputePi 127.0.0.1 45
3.141592653589793238462643383279502884197169399

C:\___Class\tmp\RMI_Example2\client>java -cp ./compute.jar;. -Djava.rmi.server.c
odebase=file:/c:/___Class/tmp/RMI_Example2/client/ -Djava.security.policy=client
.policy ComputePi 127.0.0.1 150
3.141592653589793238462643383279502884197169399375105820974944592307816406286208
998628034825342117067982148086513282306647093844609550582231725359408128

C:\___Class\tmp\RMI_Example2\client>
```

The API Documentation

Karoly.Bosa@jku.at

API = Application Programming Interface

- 166 packages
- 3417 classes
- > 35000 methods and data fields

Online Documentation (JavaDoc):

http://java.sun.com/javase/6/docs/api/

For downloading, the link is located on the “J2SE 6.0 Documentation”:

http://java.sun.com/javase/downloads/index.jsp