Distributed programming with Java

Distributed programming with Java

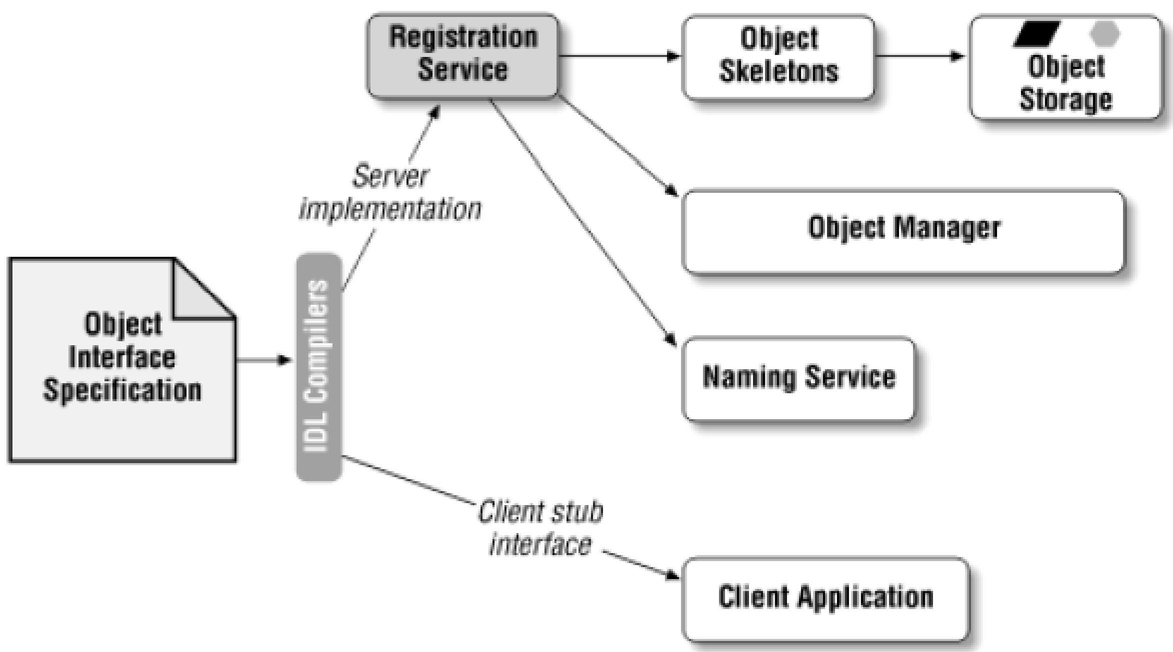
- Introduction
- RMI
- Serialization
- The Reflection API
- Case study – mobile agents

Introduction

- *Distributed programming* is the process of breaking down an application into individual computing objects that can be scattered across a network of computers, yet still work together to do cooperative tasks
- Main advantages of distributed computing include:
  - *Parallelism*: using smaller, cheaper computers to solve large problems instead of resorting to large computers
  - *Reduced network bandwidth*: large data sets are typically difficult to relocate, and easier to control and administer where they are, so remote data servers can be used to provide needed information
  - *Redundancy*: computing objects on multiple networked computers can be used by systems that need fault-tolerance; if a machine goes down, the job can still carry on

Distributed object systems

- *Distributed object system* (DOS) represents a set of APIs that hides the complexity of distributed programming
- It allows programmers to invoke objects on remote hosts, and interact with them as if they were objects within the local host
- A general architecture of a distributed object system:



DOS features

- *Object interface specification*: provides the means (e.g. a language) for specifying object interfaces, regardless of the implementation details
- *Stubs*: responsible for dispatching remote requests, stubs are used to route local method invocations to the object on a server
- *Skeletons*: responsible for processing remote requests, skeletons are used by servers to create new instances of remote objects and to route remote method calls to the object implementation
- *Naming service*: associates a remote object with a name that clients can use to obtain a reference to the object

DOS features (cont')

- *Object manager*: the heart of the distributed object system, it manages the object skeletons and object references on an object server
  - Usually, it also supports more advanced features, such as object persistence
- *Registration service*: registers newly implemented classes with a naming service and an object manager, and then stores the class in the server's storage
- *Object communication protocol*: supports the means for transmitting and receiving object references, method references, and data
  - Ideally, the client application does not know any details about this protocol

Distributed programming with Java

- Introduction
- RMI
- Serialization
- The Reflection API
- Case study – mobile agents

RMI

- *Remote Method Invocation* (RMI) is a core Java API and class library that allows Java programs running in one JVM to call methods in objects running in a different, possibly physically distributed, JVM
  - RMI creates the illusion that this distributed program is running on one system with one memory space
- It is the pure Java approach to distributed software development
- For its functionality, RMI relies on advanced Java features, such as *object serialization*, *reflection*, and *dynamic class loading*
- The RMI API is available under the *java.rmi* package

RMI services

- A *RMI service* is a remote object with methods that may be invoked from a different JVM than the one in which the object itself lives
- Each RMI service implements the *remote interface* that specifies which of its methods can be invoked by clients
- From the programmer's perspective, RMI services work pretty much like the local objects
  - That is, clients invoke the methods of the remote object almost exactly as they invoke local methods
- However, RMI is much slower and less reliable than regular local method invocation
  - Things can and do go wrong with RMI that do not affect local method invocations

The RMI registry

- In order to allow clients to find them, RMI services must be registered with a lookup service called *RMI registry*
- The registry runs as a separate process and allows applications to register RMI services or obtain a reference to a named service
- Each registered service is assigned a name which clients use to find the service
  - The clients are unaware of the actual, physical location of the service
- Included as part of the Java platform is a RMI registry application called *rmiregistry*, which can be set up to listen for incoming connections

Defining functionality of a RMI service

- The first step in creating a RMI service is defining its functionality in an interface that implements the *Remote* interface
- *Remote* is a *marker* interface that does not have any methods of its own
    - Its sole purpose is to “tag” objects so that they can be identified as RMI services
- This sub-interface of *Remote* determines which methods of the RMI service clients may call
    - A RMI service may have many public methods, but only those declared in a remote interface can be invoked remotely
    - Other public methods may be invoked only from within the virtual machine where the object lives

Implementing a RMI service

- Besides implementing methods of the RMI service interface, the class that serves as a RMI service implementation should extend the *UnicastRemoteObject* class
- This class provides all the core RMI functionality, such as exporting a RMI service and obtaining a client that communicates with the service
- Extending from *UnicastRemoteObject* is the only RMI-specific code that needs to be written for a service implementation
  - Beyond that, there is no actual networking code required
- Once a service implementation exists, the *rmic* tool, which ships with the JDK, is used to create stubs and skeletons
  - Use of *rmic* with RMI is deprecated in newer versions of Java, since stubs and skeletons are generated dynamically

Binding RMI services

- The process of associating a RMI service with a name in a registry is called *binding*
- Binding can be performed through static methods of the *Naming* class:
  - **void bind(String name, Remote obj)**: binds the specified name to a remote object
  - **void rebind(String name, Remote obj)**: rebinds the specified name to a new remote object; any existing binding for the name is replaced
  - **void unbind(String name)**: destroys the binding for the specified name that is associated with a remote object

Binding RMI services (cont')

  - **String[] list(String registryName)**: returns an array of the names bound in the specified registry
- The name of each RMI service is specified in URL format of the form *rmi://host:port/name*
- The *host* and *port* parameters are optional: the host defaults to the local host, while to port defaults to 1099
  - The *rmi* scheme specification is also optional
- The object that actually binds and instantiates a RMI service is called a *RMI server*

Invoking a RMI service

- To invoke a RMI service, the client application needs only to obtain an object reference to the remote interface
  - It does not need to be concerned with how messages are sent or received, or where the service is located
- To find the service initially, a lookup in the RMI registry is made: the client application invokes the following static method of the *Naming* class
  - **Remote lookup(String name)**: returns a stub for the RMI service associated with the specified name

RMI example – remote interface definition

- Note that each method in the remote interface needs to declare *RemoteException* in its list of possible exceptions

```
class DivisionByZero extends Exception {
  public DivisionByZero(String reason) {
    super(reason);
  }
}

public interface Calculator extends Remote {
  double add(double x, double y) throws RemoteException;
  double subtract(double x, double y) throws RemoteException;
  double multiply(double x, double y) throws RemoteException;
  double divide(double x, double y)
    throws RemoteException, DivisionByZero;
}
```

RMI example – RMI service implementation

```
public class CalculatorImpl
  extends UnicastRemoteObject implements Calculator {
  private static final double EPSILON = 0.0001;

  // an implicit constructor that throws
  // RemoteException needs to exist
  protected CalculatorImpl() throws RemoteException { }

  @Override
  public double divide(double x, double y)
  throws RemoteException, DivisionByZero {
    if (Math.abs(y) < EPSILON)
      throw new DivisionByZero("Cannot divide " + x + " by zero");
    return x / y;
  }

  ...
}
```

- The compiled service implementation can be fed to the *rmic* tool to produce the stub and skeleton

RMI example – running the RMI server

- The role of a RMI server is to register a RMI service with the running instance of a RMI registry

```
public class CalculatorServer {
  public static void main(String[] args) {
    try {
      String host = args.length >= 1 ? args[0] : "";
      String port = args.length >= 2 ? args[1] : "";
      // build the name
      String name = String.format("//%s:%s/Calculator", host, port);
      // bind an instance of the calculator to this name
      Naming.rebind(name, new CalculatorImpl());
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

RMI example – client implementation

```
public class CalculatorClient {
  public static void main(String[] args) {
    String host = args.length >= 1 ? args[0] : "";
    String port = args.length >= 2 ? args[1] : "";
    try {
      // acquire a reference to the stub
      String name = String.format("//%s:%s/Calculator", host, port);
      Remote remote = Naming.lookup(name);
      CalculatorImpl_Stub calc = (CalculatorImpl_Stub)remote;
      // invoke some operations
      System.out.printf("%f + %f = %f\n", 5.0, 12.0, calc.add(5, 12));
      try {
        System.out.printf("%f / %f = %f\n", 5.0, 0.0, calc.divide(5, 0));
      } catch (DivisionByZero e) {
        System.out.println(e.getMessage());
      }
    } catch (Exception ex) {
      ex.printStackTrace();
    }
  }
}
```

- Console:

```
5,000000 + 12,000000 = 17,000000
Cannot divide 5.0 by zero
```

Distributed programming with Java

- Introduction
- RMI
- Serialization
- The Reflection API
- Case study – mobile agents

Serialization

- *Serialization* is the process of converting a set of object instances that contain references to each other into a linear stream of bytes
  - It is the mechanism used by RMI to pass objects between JVMs
- The serialization process of an object preserves:
  - The class name and signature of the class
  - The values of all non-static object's fields
  - The closure of any other objects referenced from the initial objects
- The reversed process – creating a set of objects from a stream of data – is called *deserialization*

Using serialization

- The three most common uses of serialization are:
  - *Persistence*: by using *FileOutputStream*, the object stream can automatically be written to a file
  - *A copy mechanism*: by using *ByteArrayOutputStream*, the object stream is written to a byte array in memory, which can then be used to create duplicates of the original object
  - *Communication*: by using a stream that comes from a socket, objects can automatically be sent over the wire to the receiving socket

Serialization requirements

- To allow an object to be serializable, its class should implement the *Serializable* interface
  - *Serializable* is a “marker” interface – it contains no elements, but simply tags the class as serializable
- A *NotSerializableException* is thrown if serialization is tried on non-serializable objects
- Serializability is inherited: it only needs to be implemented once along the class hierarchy
  - Most Java classes are serializable

The *transient* keyword

- In order to serialize an object, all of its fields have to be serializable as well
  - Serialization does not care about access modifiers, such as *private*
- To serialize a class with non-serializable fields, mark the properties with the *transient* keyword
  - The *transient* keyword prevents the data from being serialized
- During deserialization, transient properties are initialized to their default values (0, *null*, etc.)
- The keyword can also be used with serializable fields, e.g. for performance reasons

Writing and reading objects

- *ObjectOutputStream* writes primitive data types and graphs of Java objects to an output stream
- It offers the following method for writing an object to a stream:
  - **void writeObject(Object obj)**
- *ObjectInputStream* deserializes primitive data and objects previously written using an *ObjectOutputStream*
- It offers the following method for reading an object from a stream:
  - **Object readObject()**

Persistent list of numbers – example

```java
public class NumList implements Serializable {
  private List<Integer> numbers;
  private int maxNum;
  private transient int minNum;

  public NumList() {
    numbers = new ArrayList<Integer>();
    maxNum = Integer.MIN_VALUE;
    minNum = Integer.MAX_VALUE;
  }

  public void add(int num) {
    numbers.add(num);
    // remember max and min values
    if (maxNum < num)
      maxNum = num;
    if (minNum > num)
      minNum = num;
  }

  @Override public String toString() {
    return String.format("%s (min:%d; max:%d)",
      numbers, minNum, maxNum);
  }
}
```

Persistent list of numbers – example (cont')

```
public class SerializationTest {

  private static NumList load(String fileName) {
    ObjectInputStream in = null;
    try {

      File file = new File(fileName);
      if (!file.exists())
        return new NumList();
      in = new ObjectInputStream(new FileInputStream(file));
      return (NumList)in.readObject();

    } catch (Exception e) {
      return new NumList();
    } finally {
      if (in != null)
        try {
          in.close();
        } catch (IOException e) { }
    }
  }
```

Persistent list of numbers – example (cont')

```
  private static void save(NumList nums, String fileName) throws IOException {
    ObjectOutputStream out = new ObjectOutputStream(
      new FileOutputStream(fileName));
    try {
      out.writeObject(nums);
    } finally {
      out.close();
    }
  }

  public static void main(String[] args) throws IOException {
    NumList nums = load("numbers.dat");
    System.out.println("Existing data: " + nums);

    for (int i = 0; i < 3; i++)
      nums.add((int)(Math.random() * 9) + 1);

    System.out.println("After adding: " + nums);
    save(nums, "numbers.dat");
  }
}
```

- First run:
```
Existing data: [] (min:2147483647; max:-2147483648)
After adding: [7, 5, 2] (min:2; max:7)
```
- Second run:
```
Existing data: [7, 5, 2] (min:0; max:7)
After adding: [7, 5, 2, 9, 4, 2] (min:0; max:9)
```

Version control

- All serializable classes are automatically given a version identifier
- This identifier is saved along with the object and automatically updated whenever the object's class changes
  - E.g. when a new field is added
- Version identifiers are compared during deserialization: if the version of the class does not equal the version of the object in the stream, an exception is thrown
- To control the versioning system, developers simply need to provide the static *serialVersionUID* field manually and ensure it is always the same, unless such changes are made to the class which invalidate previously serialized objects

Distributed programming with Java

- Introduction
- RMI
- Serialization
- The Reflection API
- Case study – mobile agents

The Reflection API

- *Reflection* is the process by which software can observe and modify (its own) program structure and behavior at runtime
- It allows inspection of classes and their elements, as well as instantiation of new objects and invocations of methods at runtime without knowing the actual names at compile time
- In Java, the access to this *object metadata* is available through an immutable instance of *java.lang.Class*
- The Java Reflection API, available under the *java.lang.reflect* package, is most commonly used by:
    - Serialization and RMI
    - Class browsers and visual development environments
    - Debuggers and test tools that, for example, need to access private properties of a class

Retrieving the object metadata

- There are several ways of retrieving the *Class* instance:
  - If an instance of an object is available, the simplest way to get its *Class* is to invoke its *getClass()* method
  - If the type is available but there is no instance, then it is possible to obtain a *Class* by appending *".class"* to the name of the type; this approach also works for primitive data types
  - If the fully-qualified name of a class is available, it is possible to get the corresponding *Class* using the static method *Class.forName()*

- For primitive types, e.g. *int*, use *int.class*, or Integer.TYPE

Examining class modifiers and types

- The following example demonstrates how to examine class modifiers, generic type parameters, inheritance path, and annotations

```
public class ClassDeclarationSpy {
  public static void main(String[] args) {
    try {
      Scanner scanner = new Scanner(System.in);
      out.print("Class name? ");
      String name = scanner.nextLine();

      Class<?> c = Class.forName(name);
      out.println("Modifiers:\n\t" +
        Modifier.toString(c.getModifiers()));

      out.println("Type Parameters:");
      TypeVariable<?>[] tv = c.getTypeParameters();
      if (tv.length == 0)
        out.println("\tNo Type Parameters");
      else
        for (TypeVariable<?> t : tv)
          out.println("\t" + t.getName());
```

Examining class modifiers and types (cont’)

```
    out.println("Implemented interfaces:");
    Type[] intfs = c.getGenericInterfaces();
    if (intfs.length == 0)
      out.println("\tNo Implemented Interfaces");
    else
      for (Type intf : intfs)
        out.println("\t" + intf);

    out.println("Inheritance path:");
    printAncestors(c);

    out.println("Annotations:");
    Annotation[] ann = c.getAnnotations();
    if (ann.length == 0)
      out.println("\tNo Annotations");
    else
      for (Annotation a : ann)
        out.print("\t" + a);

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

Examining class modifiers and types (cont’)

```
  private static void printAncestors(Class<?> c) {
    Class<?> ancestor = c.getSuperclass();
    if (ancestor != null) {
      out.println("\t" + ancestor.getCanonicalName());
      printAncestors(ancestor);
    }
  }
}
```

- Console:

```
Class name? java.lang.String;
Modifiers:
  public abstract final
Type Parameters:
  No Type Parameters
Implemented interfaces:
  interface java.lang.Cloneable
  interface java.io.Serializable
Inheritance path:
  java.lang.Object
Annotations:
  No Annotations
```

Discovering class members

- There are two categories of methods provided in *Class* for accessing fields, methods, and constructors:
  - Methods which enumerate all members
  - Methods which search for a particular member
- Additionally, there are distinct methods for accessing members declared directly on the class versus methods which search the super-interfaces and super-classes for inherited members
- Finally, some methods of *Class* can look for public members only, while other can also access private and protected members

Discovering class members – example

- The following example demonstrates how to list all members of a class, declared directly on the class

```
package test;

public class SampleClass {
  private class InnerClass {
    private int k;
    private InnerClass(int k) { this.k = k; }
  }

  private int privateField;
  public InnerClass publicField;

  protected SampleClass() { }

  public SampleClass(int n) {
    privateField = n;
    publicField = new InnerClass(n);
  }

  public int getValue() {
    return privateField + publicField.k;
  }
}
```

Discovering class members – example (cont')

```
public class ClassMemberSpy {
  private static void printClass(Class<?> c) {
    out.println("Class: " + c.getName());
    out.println("Package: " + c.getPackage());

    printMembers("Constructors:", c.getDeclaredConstructors());
    printMembers("Methods:", c.getDeclaredMethods());
    printMembers("Fields:", c.getDeclaredFields());

    out.println("Inner classes:");
    Class<?>[] inner = c.getDeclaredClasses();
    if (inner.length == 0)
      out.println("\tNone");
    else
      for (Class<?> cls : inner)
        printClass(cls);
  }

  private static void printMembers(String msg, Member[] members) {
    out.println(msg);
    if (members.length == 0)
      out.println("\tNone");
    else
      for (Member m : members)
        out.println("\t" + m);
  }
```

Discovering class members – example (cont’)

```
  public static void main(String[] args) throws ClassNotFoundException {
    Scanner scanner = new Scanner(System.in);
    out.print("Class name? ");
    String name = scanner.nextLine();
    Class<?> c = Class.forName(name);
    printClass(c);
  }
}
```

- Console:

```
Class name? test.SampleClass
Class: test.SampleClass
Package: package test
Constructors:
  protected test.SampleClass()
  public test.SampleClass(int)
Methods:
  public int test.SampleClass.getValue()
Fields:
  private int test.SampleClass.privateField
  public test.SampleClass$InnerClass test.SampleClass.publicField
Inner classes:
Class: test.SampleClass$InnerClass
Package: package test
Constructors:
  ...
```

Distributed programming with Java

- Introduction
- RMI
- Serialization
- The Reflection API
- Case study – mobile agents

Case study – mobile agents

- *Agent technology* represents one of the most consistent approaches to distributed software development
- It employs a distributed network of autonomous, executable software entities called *agents*
- An agent is considered to be *mobile* if it can move from one computer in a network to another
- A common use of mobile agents is in processing of large data sets:
  - Retrieving a large data set from a remote database to a computer hosting the processing software can be too expensive, or even impossible
  - A much better approach is to send a (usually small) agent to the target computer and perform the processing on-the-spot

Agent server

- Each computer that wants to accept mobile agents hosts an *agent server*
- The server accepts a serialized form of a mobile agent, and then deserializes and runs it
- To make the server as general as possible, the Reflection API is used to run the agent
  - The server will look for and invoke the following method: *void onArrival(ServerSocket host)*

Agent server implementation

```
public class AgentServer extends Thread {
  public static final int PORT = 6060;

  @Override public void run() {
    ServerSocket socket = null;
    try {
      socket = new ServerSocket(PORT);
      while (true) {
        Socket client = socket.accept();
        new AgentHandler(socket, client).start();
      }
    } catch (Exception ex)
    {
      ex.printStackTrace();
    } finally
    {
      if (socket != null)
        try
        {
          socket.close();
        } catch (IOException e) { }
    }
  }
}
```

Agent server implementation (cont')

```
public class AgentHandler extends Thread {
  private ServerSocket server;
  private Socket client;

  public AgentHandler(ServerSocket server, Socket client) { ... }

  @Override public void run() {
    ObjectInputStream in = null;
    try {
      in = new ObjectInputStream(client.getInputStream());
      // deserialize the agent and get its class
      Object agent = in.readObject();
      Class<?> c = agent.getClass();
      // look for the "onArrival" method
      // with one parameter of type ServerSocket
      Method m = c.getMethod("onArrival", ServerSocket.class);
      // invoke the method
      m.invoke(agent, server);
    } catch (Exception e) {
      e.printStackTrace();
    } finally {
      if (in != null)
        try {
          in.close();
        } catch (IOException e) { }
    }
  }
}
```

The mobile agent

- The source code of the mobile agent is very simple:

```java
public class MobileAgent implements Serializable {
  public void moveTo(String host, int port) {
    Socket socket = null;
    ObjectOutputStream out = null;
    try {
      socket = new Socket(host, port);
      out = new ObjectOutputStream(socket.getOutputStream());
      out.writeObject(this);
    } catch (Exception e) {
      e.printStackTrace();
    } finally {
      // close socket and output stream
    }
  }

  public void onArrival(ServerSocket host) {
    // perform data processing on the current host
  }
}
```