Example15

Table of contents

Aspect-Oriented Programming

Remote Aspect

Un oggetto si dice **remoto** se i suoi servizi possono essere utilizzati da oggetti che vivono su altre macchine sulla rete. Avremo la necessita' di trasferire informazioni avanti e indietro e trasformare lo scambio dei messaggi in visione orientata agli oggetti.

Sarebbe il predisposto di base per la realizzazione dei *sistemi distribuiti*: le parti di un programma lavorano in macchine diverse collegate da rete.
Ci servono:

- meccanismo per spostare dati sulla rete → se un oggetto implementa `Serializable` questo puo' essere trasformato in array di byte;
- *socket* → file descriptor per read/write, di tipo TCP

In tutto questo meccanismo, i proxy ci servono sul mittente: siccome vogliamo nascondere l'invio del messaggio, costruiremo proxy che mascherano la chiamata remota, prendono gli argomenti, impacchettano, inviano e aspettano il risultato e una volta ottenuto si sbloccano.

Per ottenere il nostro socket, ci serve sapere prima se siamo su lato *server*, aspettando la richiesta di connessione, o sul lato *client*, se facciamo la richiesta accettata o rifiutata che sia. 2 classi diverse nel caso di richiesta:

- l'oggetto server usa `java.net.ServerSocket`

  una volta costruito, mediante la chiamata `accept()` possiamo metterci in attesa di connessione; stabilita una *porta* (tra 0 e 64K) che venga utilizzata per aspettare connessioni,

che non sia gia' in uso (solitamente quelle sotto 1024 sono riservate); a connessione stabilita, viene creato un oggetto socket dello stesso tipo del client; ogni connessione va elaborata in un thread pool.

- l'oggetto client usa `java.net.Socket`

  mediante la costruzione oggetto socket, possiamo fare una richiesta di connessione; se va a buon fine abbiamo input/output stream per inviare e ricevere dati su canale; un proxy riceve gli argomenti della chiamata, istituisce connessione con socket server, utilizzi client inviando dati; l'*indirizzo* della macchina e la porta vengono passate al costruttore.

`it.unipr.informatica.aspects`

`RemoteRequest.java`

La richiesta, anziche' fare un oggetto subito, facciamo un'interfaccia.
Ci permettera' di avere richieste/risposte su mittente/ricevente con classi diverse.

- nome del metodo che stiamo richiedendo, quello che si trova dopo il punto nel fare la chiamata;
- valori degli argomenti, quali sono gli oggetti che stiamo passando nella richiesta;
- i tipi dei parametri che stanno identificando la richiesta.

```
public interface RemoteRequest extends Serializable {
        public String getMethodName();
        public String[] getParameterTypeNames();
        public Object[] getArguments();
}
```

`RemoteResponse.java`

Una volta costruita la richiesta, andiamo a produrre la risposta.

- oggetto che e' il risultato;
- eccezione a seconda se sia stato generato un oggetto o eccezione.

```
public interface RemoteResponse extends Serializable {
        public Object getResult();
        public Throwable getException();
}
```

`RemoteException.java`

Siccome i tipi delle eccezioni che ci potrebbero essere ritornate, possono essere dei piu' variegati, ci conviene creare un'interfaccia.

Tutte le volte che il nostro socket-client non riuscira' a collegarsi perche' in quel momento la rete e' down, prenderemo un'eccezione che sara' causa della nostra eccezione remota, lanceremo questa e chi ha fatto la richiesta verra' informato.

Siccome siamo partiti dal presupposto che non si conosca del fatto che i nostri non sono oggetti remoti, lanciamo una `RuntimeException`.

```
public class RemoteException extends RuntimeException {
        private static final long serialVersionUID = -587659957946875703L;
        public RemoteException(Throwable cause) {
                super(cause);
        }
}
```

`RemoteHandler.java`

Il remote aspect generera' un handler.
Ha l'unico scopo di aggiungere tutte le funzionalita' tipiche dell'aspetto: passare dall'handler rendiamo evidente che l'oggetto che cerchiamo di manipolare ha quel certo aspetto. Il remote handler viene costruito sul server: una volta costruito, questo costruisce pool di thread, va a prendere la porta TCP/IP mettendosi in attesa di richieste e accettandole con `accept()`.

Ci serve un modo per fare `shutdown()` siccome la cosa diventa complicata e non vogliamo costruire il pool se tanto non andiamo avanti per eccezioni: nel remote handler abbiamo un solo metodo che ha compito di prendere l'oggetto reso remoto e chiuderlo.

```
public interface RemoteHandler<T> {
        public void shutdown();
}
```

Nel lato client l'handler non serve, quello che avremo sara' invece un proxy del tutto trasparente: implementera' interfaccia `T`, lo useremo come se fosse locale.

`RemoteAspect.java`

La richiesta avra':

- nome di un metodo;
- elenco di nomi di tipi di parametri;
- elenco di oggetti che funzionano d'argomenti.

> ⚠ **Differenza *argomenti* e *parametri***
>
> L'*argomento* e' il valore a tempo d'esecuzione,
> il *parametro* e' il nome che utilizziamo a tempo d'esecuzione e inoltre ha anche un tipo (tipico dei linguaggi staticamente tipati come JAVA  .

```
        // ...
        private static class InnerRemoteRequest implements RemoteRequest {
                private static final long serialVersionUID =
                        7994546295394535576L;
                private String methodName;
                private String[] parameterTypeNames;
                private Object[] arguments;
```

```
                private InnerRemoteRequest
                (String methodName, String[] parameterTypeNames,
                        Object[] arguments) {
                        this.methodName = methodName;
                        this.parameterTypeNames = parameterTypeNames;
                        this.arguments = arguments;
                }

                @Override
                public String getMethodName() {
                        return methodName;
                }

                @Override
                public String[] getParameterTypeNames() {
                        return parameterTypeNames;
                }

                @Override
                public Object[] getArguments() {
                        return arguments;
                }
        }
        // ...
```

Per la risposta ci serve:

- un oggetto che necessariamente deve essere serializzabile che e' il risultato;
- un eccezione, tutti i throwable sono serializzabili.

E' privata, siccome l'implementazione e' nostro problema, e' statica siccome non vogliamo che dipenda in nessun modo allo stato dell'aspetto.

```
        // ...
        private static class InnerRemoteResponse
        implements RemoteResponse {
                private static final long serialVersionUID =
                        8159305632457402638L;
                private Serializable result;
                private Throwable exception;

                public InnerRemoteResponse
                        (Serializable result, Throwable exception) {
                        this.result = result;
                        this.exception = exception;
                }

                @Override
                public Serializable getResult() {
                        return result;
                }

                @Override
```

```
                public Throwable getException() {
                        return exception;
                }
        }
}
```

Per mandare le richieste utilizziamo un proxy che viene ritornato dalla chiamata `connect()`.
Costruisce un proxy che permette l'invio di richieste e l'attesa della risposta e ha bisogno:

- descrittore dell'interfaccia che andiamo a implementare;
- nome dell'host su cui andiamo a fare collegamento (DNS o IPv4 o simili);
- numero della porta (1 → 64K), identico a quello dell'altra parte, il server.

Gli argomenti vengono verificati e il proxy costruito.
Riceve le chiamate e impacchetta tutto in una richiesta, apre una connessione verso il server, inviera' richiesta e si mette in attesa e una volta che la risposta arriva, spacchetta e ritorna. Di questo se ne occupa l'invokation handler.

```
        // ...
        public static <T> T connect
                (Class<T> remoteInterface, String host, int port)
                throws IOException {
                if (remoteInterface == null)
                        throw new IllegalArgumentException
                                ("remoteInterface == null");

                if (host == null || host.length() == 0)
                        throw new IllegalArgumentException
                                ("host == null || host.length() == 0");

                if (port < 1 || port > 65535)
                        throw new IllegalArgumentException
                                ("port < 1 || port > 65535");

                @SuppressWarnings("unchecked")
                T result = (T) Proxy.newProxyInstance
                (remoteInterface.getClassLoader(),
                        new Class<?>[] { remoteInterface },
                                new InnerInvocationHandler(host, port));

                return result;
        }
        // ...
```

L'invokation handler ha bisogno:

- dell'host a cui fare collegamento, fatto nel momento della chiamata;
- della porta.

Nel momento in cui arriva una chiamata, finiamo nel metodo `invoke()` dove ci servono i pezzi per costruire la richiesta che comprende:

- il nome;

- i nomi dei tipi dei parametri;
- gli argomenti.

Anziche' inviare i tipi dei parametri, ovvero un sacco d'informazioni in piu', tiriamo fuori da questi oggetti di tipo `Class`, i nomi.
Costruita la request, costruiamo il socket e siccome siamo nel lato client ci basta passare come argomenti l'host e la porta.
Fatto cio', se viene ritornato, vuole dire che il collegamento e' stato stabilito: nel momento in cui la `new Socket(host, port` ritorna, vuol dire che tutto e' andato a buon fine, altrimenti eccezione che una volta catturata lancia `RemoteException`.

Costruito il socket, ci facciamo dare l'input stream (da cui leggiamo) e l'output stream (su cui scriviamo) per la request: prendere l'oggetto `request`, trasformarlo in array di byte e mandarlo.
L'oggetto `OutputStream` permette di scrivere oggetti.
L'oggetto `InputStream`, all'apertura, prima cosa che fa e' mettersi in attesa dell'header dei dati e tramite `readObject()`, appena arriva, legge.
L'oggetto e' una `RemoteResponse` vuole dire che abbiamo ottenuto risposta alla richiesta, se non lo e' vuole dire che il server ha ritornato altro.

- se dentro alla risposta c'e' eccezione, la rilanciamo;
- se c'e' dentro il risultato, lo ritorniamo direttamente al proxy.

```
        // ...
        private static class InnerInvocationHandler
        implements InvocationHandler {
                private String host;
                private int port;
                private InnerInvocationHandler(String host, int port) {
                        this.host = host;
                        this.port = port;
                }

                @Override
                public Object invoke
                (Object proxy, Method method, Object[] arguments)
                throws Throwable {
                        String methodName = method.getName();
                        Class<?>[] parameterTypes = method.getParameterTypes();
                        int parameterCount = parameterTypes.length;
                        String[] parameterTypeNames = new String[parameterCount];

                        for (int i = 0; i < parameterCount; ++i)
                                parameterTypeNames[i] = parameterTypes[i].getName();

                        RemoteRequest request =
                        new InnerRemoteRequest
                        (methodName, parameterTypeNames, arguments);

                        RemoteResponse response = null;

                        try (Socket socket = new Socket(host, port);
                                        InputStream inputStream =
```

```
                                        socket.getInputStream();
                                        OutputStream outputStream =
                                        socket.getOutputStream();
                                        ObjectOutputStream objectOutputStream =
                                        new ObjectOutputStream(outputStream);) {

objectOutputStream.writeObject(request);

                                                try (ObjectInputStream
objectInputStream =
                                                new ObjectInputStream(inputStream);)
{
                                                Object message =
                                                objectInputStream.readObject();


                                                if (!(message instanceof
RemoteResponse))
                                                        throw new
IllegalArgumentException
                                                        ("!(message instanceof
RemoteResponse)");

                                                response = (RemoteResponse) message;
                                }
                        } catch (Throwable throwable) {
                                throw new RemoteException(throwable);
                        }

                        Throwable exception = response.getException();
                        if (exception ≠ null)
                                throw exception;
                        return response.getResult();
                }
        }
        // ...
```

Sul server le cose sono piu' complicate siccome dobbiamo passare dall'handler.
Sta volta `attach()` viene utilizzata per attaccare un aspetto remoto a un oggetto che abbiamo gia' a disposizione e per renderlo accessibile quello che faremo sara':

- prendere un oggetto qualsiasi;
- istanziare l'oggetto;
- attaccare l'aspetto remoto dicendo, quale e' la sua interfaccia per renderlo remoto, quale e' la porta per metterci in attesa delle connessioni + argomento opzionale il numero di pool di thread.

Verifichiamo la validita' degli argoementi e costruiamo un server-socket con bind della porta. Una volta che abbiamo la porta costruiamo il remote handler `InnerRemoteHandler<T>`.

```
        // ...
        public static <T> RemoteHandler<T> attach
        (Class<T> remoteInterface, T target, int port, int poolSize)
                throws IOException {
```

```
                if (remoteInterface == null)
                        throw new IllegalArgumentException
                        ("remoteInterface == null");

                if (target == null)
                        throw new IllegalArgumentException
                        ("target == null");

                if (port < 1 || port > 65535)
                        throw new IllegalArgumentException
                        ("port < 1 || port > 65535");

                ServerSocket serverSocket = new ServerSocket(port);

                return new InnerRemoteHandler<T>
                (serverSocket, target, poolSize);
        }
        // ...
```

L'handler costruisce l'esecutore scegliendo come pool di thread fissato a numero di thread + 1 (ci sara' sempre uno dei thread impegnato all'attesa di connessione).

```
        // ...
        private static class InnerRemoteHandler<T>
        implements RemoteHandler<T> {
                private Object target;
                private ExecutorService executorService;
                private ServerSocket serverSocket;

                private InnerRemoteHandler
                (ServerSocket serverSocket, Object target, int poolSize) {
                        this.serverSocket = serverSocket;
                        this.target = target;
                        this.executorService =
                                Executors.newFixedThreadPool(poolSize + 1);
                        executorService.execute(this::serverLoop);
                }
                // ...
```

Ci basta fare `shutdown()` sul pool di thread, facendo si che uno dopo l'altro finiscano di elaborare e terminino, chiudendo anche il `serverSocket`.

```
                // ...
                @Override
                public void shutdown() {
                        executorService.shutdown();

                        try {
                                serverSocket.close();
                        } catch (Throwable e) {
                                // Blank
                        }
                }
                // ...
```

Prende il `serverSocket` e chiama `accept()` in ciclo forever.
Questa si sblocca per 2 motivi:

1. qualcuno ha chiuso il socket, e' stato fatto `shutdown()`, quindi usciamo;
2. e' arrivata una richiesta, ritorniamo il socket utilizzato per la richiesta.

Il socket ritornato dall'`accept()` serve quindi a ritornare i due estremi che in mezzo hanno la rete.

```
                // ...
                private void serverLoop() {
                        try {
                                for (;;) {
                                        Socket socket = serverSocket.accept();
                                        executorService.execute(() →
serve(socket));
                                }
                        } catch (SocketException exception) {
                                // Blank
                        } catch (Throwable throwable) {
                                throwable.printStackTrace();
                        }
                }
                // ...
```

`serve()` non la chiamiamo direttamente, costruiamo invece un task che viene eseguito dall'esecutore nei suoi thread e che prende il socket costruito da `accept()` e lo usa per:

- ricevere una richiesta;
- spacchettare la richiesta;
- cercare il metodo da invocare, invocare tramite `Reflection`, ottenendo risultato o eccezione che poi verra' mandata indietro.

```
                // ...
                private void serve(Socket socket) {
                        try (InputStream inputStream = socket.getInputStream();
                                        OutputStream outputStream =
                                                socket.getOutputStream();
                                        ObjectInputStream objectInputStream =
                                                new ObjectInputStream(inputStream);
                                        ObjectOutputStream objectOutputStream =
                                                new
ObjectOutputStream(outputStream);) {
                                Object message = objectInputStream.readObject();

                                if (!(message instanceof RemoteRequest))
                                        throw new IllegalStateException
                                        ("!(message instanceof RemoteRequest)");

                                RemoteRequest request = (RemoteRequest) message;
                                String methodName = request.getMethodName();
                                String[] parameterTypeNames =
                                        request.getParameterTypeNames();
                                Class<?>[] parameterTypes =
```

```
                                        new Class[parameterTypeNames.length];

                                for (int i = 0; i < parameterTypeNames.length; ++i)
                                        parameterTypes[i] =

getClassFromName(parameterTypeNames[i]);

                                Class<?> targetClass = target.getClass();
                                Method method = targetClass.getMethod
                                        (methodName, parameterTypes);
                                RemoteResponse response;

                                try {
                                        Object result = method.invoke
                                                (target, request.getArguments());

                                        if (!(result instanceof Serializable))
                                                throw new IllegalStateException
                                                ("!(result instanceof
Serializable)");

                                        response = new InnerRemoteResponse
                                                ((Serializable) result, null);
                                } catch (InvocationTargetException exception) {
                                        Throwable throwable = exception.getCause();

                                        if (!(throwable instanceof Serializable))
                                                throw new IllegalStateException
                                                ("!(throwable instanceof
Serializable)");

                                        response = new InnerRemoteResponse
                                                (null, throwable);
                                }
                                objectOutputStream.writeObject(response);
                                socket.close();
                        } catch (Throwable throwable) {
                                throwable.printStackTrace();
                        }
                }
                // ...
```

Example15

Due sono gli eseguibili che abbiamo: l'eseguibile che costruisce un oggetto e lo rende accessibile da remoto (server), l'eseguibile che si collega al server e richiede i servizi (client).

`FileManager.java`

Interfaccia esempio.
Possiamo chiedergli di dare l'elenco dei file contenuti in una cartella con ritorno un elenco di nomi dei file, e dare il loro contenuto in array di byte.
L'importante e' il ritorno del nome di file e non cartelle.

```java
public interface FileManager {
	public String[] listFileNames(String folderName)
		throws IOException;
	public byte[] getFile(String path) throws IOException;
}
```

`SimpleFileManager.java`

Oggetto da rendere remoto.
`listFileNames()` prende il nome di una cartella e utilizzando `listFiles()` sull'oggetto costruito dal nome, va a elencare tutto.
Per ogni file, prende il nome e lo mette in una lista che poi va ritornata.

`Example15Server.java`

Costruisce un `simpleFileManager` specificando una cartella di base.
Tutte le operazioni vanno verso la cartella di base.
L'aspetto logging viene aggiunto per tracciare tutti i metodi inviati in esecuzione, utile per scrivere gli accessi remoti che arrivano.

```java
public class Example15Server {
	private void go() {
		FileManager fileManager = new SimpleFileManager
			("src/it/unipr/informatica/examples");
		fileManager = LoggingAspect.attach(fileManager);

		try {
			RemoteHandler<FileManager> fileManagerHandler =
				RemoteAspect.attach
				(FileManager.class, fileManager, 1704);
			Thread.sleep(60000);
			fileManagerHandler.shutdown();
		} catch (Throwable throwable) {
			throwable.printStackTrace();
		}
	}

	public static void main(String[] args) {
		new Example15Server().go();
	}
}
```

`Example15Client.java`

Facciamo `connect()` dicendo:

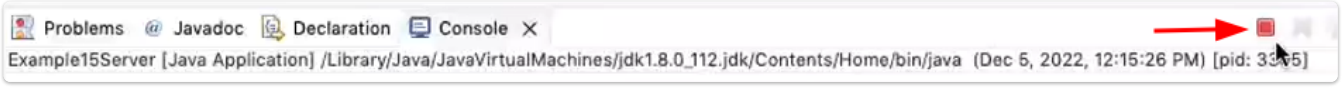
- quale e' l'interfaccia remota;
- il nome della macchina;
- la porta.

Stiamo dicendo con localhost, che il servizio remoto e' sulla nostra macchina.
Facciamo `listFileNames()` di `.`, elencando il contenuto del base path.

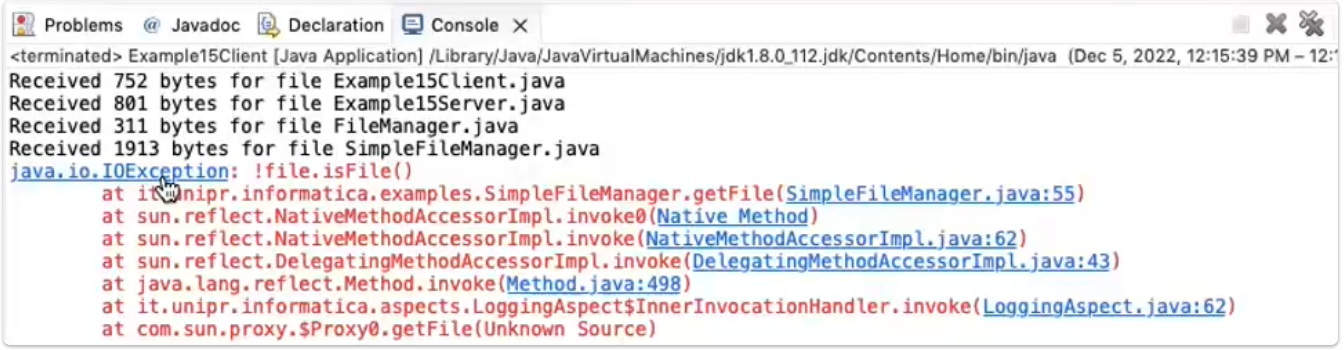
Otterremo un array di byte la cui lunghezza verra' stampata.
Facciamo anche il caso di un file non esistente per mandare eccezione.

```
public class Example15Client {
        private void go() {
                try {
                        FileManager fileManager = RemoteAspect.connect
                                (FileManager.class, "127.0.0.1", 1704);
                        String[] fileNames = fileManager.listFileNames(".");

                        for (String fileName : fileNames) {
                                byte[] file = fileManager.getFile(fileName);
                                System.out.println
                                        ("Received " + file.length +
                                        " bytes for file " + fileName);
                        }

                        fileManager.getFile("missingFile");
                } catch (Throwable throwable) {
                        throwable.printStackTrace();
                }
        }

        public static void main(String[] args) {
                new Example15Client().go();
        }
}
```

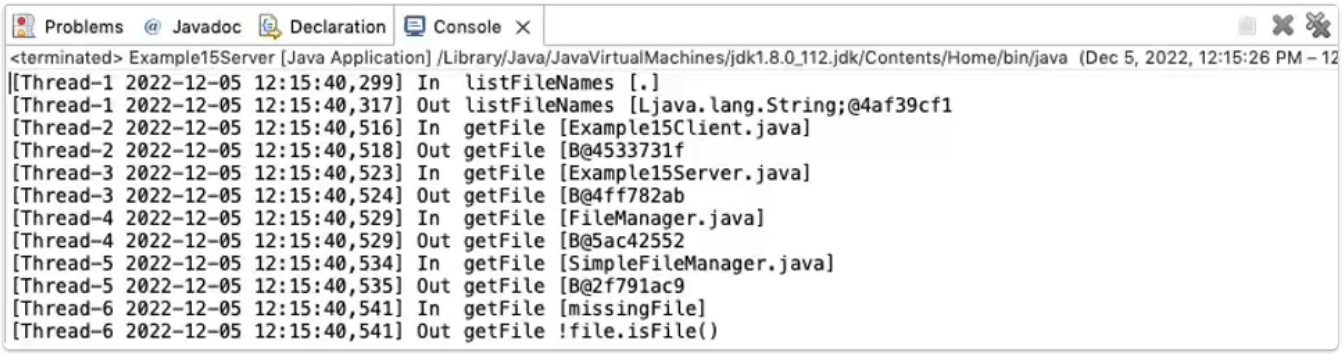
Il server viene fatto partire.



Parte anche il client che fa richieste al server.



Tramite il logging aspect vengono stampati, sul server, gli accessi.



Se la porta non e' utilizzata, e quindi nessuno accetta connessioni, viene ritornata 'Connection refused'.