

# **Laboratorio di Programmazione di Rete**

## **Corso A**

*Anno Accademico 2005-2006*

*Docente: Laura Ricci*

*Assistente: Paolo Mori*

# INFORMAZIONI UTILI

## *Orario del Corso:*

Lunedì 9.00 -11.00 – Laboratorio Gruppo A

11.00-13.00 – Laboratorio Gruppo B

Mercoledì 9.00 -11.00 – Lezione, Aula A

*Laboratorio:* Verifica degli esercizi assegnati nella lezione teorica precedente. (verrà controllata la correttezza degli esercizi svolti da alcuni gruppi estratti a sorte)

Orario di Ricevimento : *Giovedì ore 15.00 – 18.00* (possibili spostamenti nel corso del semestre)

Modalità di esame: Progetto finale + Orale

Orale: discussione del progetto + domande sugli argomenti trattati nelle lezioni teoriche (soprattutto quelli non coperti dal progetto)

# INFORMAZIONI UTILI

Prerequisiti: conoscenza del linguaggio JAVA, in particolare definizione di packages, gestione delle eccezione, multithreading, streams

Linguaggio di programmazione: *JAVA 1.5*, librerie *JAVA.NET*, *JAVA.RMI*.

Ambiente di Sviluppo: Linux, Eclipse 1.3 oppure emacs+JAVAC +JAVA

Materiale Didattico:

- Lucidi delle lezioni
- M.L. Liu – *Distributed Computing, Principles and Applications*, Addison Wesley
- Harold - *JAVA Network Programming* 2nd edition O'Reilly 2001 (la versione tradotta in italiano riguarda la prima edizione del testo).

Materiale di Consultazione:

M.Hughes, M. Shoffner, D.Hammer – *Java Network Programming*

Per i costrutti di base, Cay Horstmann – *Concetti di Informatica e Fondamenti di Java2*

# PROGRAMMA DEL CORSO

*Meccanismi per la Gestione delle Eccezioni:* Sollevamento, Rilevazione, Gestione di eccezioni. Costrutti throws, try-catch.

*Threads:* Classe Threads, Interfaccia Runnable, Attivazione, Priorità, Strutture dati condivise, Metodi synchronized.

*Streams:* Proprietà degli Streams, Tipi di streams, Composizione di streams, ByteArrayInputStream, ByteArrayOutputStream.

*Indirizzamento IP:* Classi di Indirizzi, Subnetting, Subnet Masks , Gestione degli Indirizzi IP in JAVA: La classe InetAddress.

*Meccanismi di Comunicazione in Rete:* Sockets Connectionless e Connection Oriented.

*Connectionless Sockets:* La classe Datagram Socket: creazione di sockets, generazione di pacchetti, timeouts, uso degli streams per la generazione di pacchetti di bytes, invio di oggetti su sockets connectionless.

*Multicast:* La classe MulticastSocket, Indirizzi di Multicast, Associazione ad un gruppo di multicast. Proposte di reliable multicast (FIFO multicast, causal multicast, atomic multicast).

# PROGRAMMA DEL CORSO

*Multicast:* La classe MulticastSocket, Indirizzi di Multicast, Associazione ad un gruppo di multicast. Proposte di reliable multicast (FIFO multicast, causal multicast, atomic multicast).

*Connection Oriented Sockets:* Le classi ServerSocket e Socket. Invio di oggetti su sockets TCP.

*Il Paradigma Client/Server:* Caratteristiche del paradigma client/ server, Meccanismi per l'individuazione di un servizio, architettura di un servizio di rete.

*Oggetti Distribuiti:* Remote Method Invocation, Definizione di Oggetti Remoti, Registrazione di Oggetti, Generazione di Stub e Skeletons.

*Meccanismi RMI Avanzati:* Il meccanismo delle callback.

*Cenni a CORBA*

# GESTIONE DELLE ECCEZIONI

*Perché la Gestione delle eccezioni?*

Si richiede di *gestire correttamente* le numerose eccezioni sollevate dai metodi delle classi della *libreria JAVA.NET* (es: localizzazione di un host remoto impossibile, spedizione di un messaggio sulla rete non eseguita con successo...).

*Passi fondamentali per una corretta gestione delle eccezioni:*

Specificazione e Lancio della Eccezione (clausole *throw + throws*)

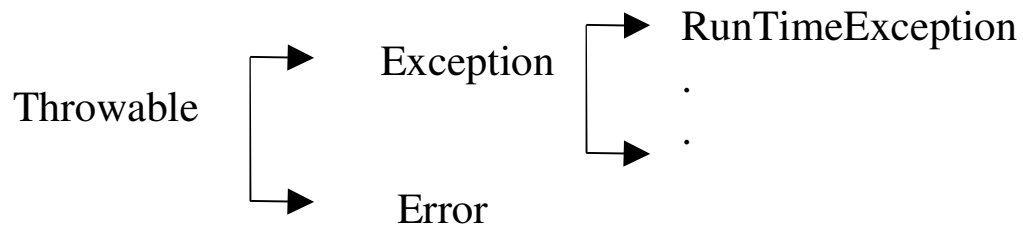
Intercettazione (clausole *try + catch*)

Gestione

Cosa è una eccezione?

È un oggetto appartenente ad una qualunque classe derivata dalla *Throwable*.

L'oggetto descrive la eccezione che è stata sollevata



# GESTIONE DELLE ECCEZIONI: SPECIFICA E LANCIO

*DivideByZeroTest* :effettua la divisione tra due numeri e solleva una eccezione se il denominatore è uguale a 0.

```
public class DivideByZeroTest
{public static double quotient (int numerator, int denominator) throws ArithmeticException
    {if (denominator == 0)
        throw new ArithmeticException ("hai diviso per 0");
    return (double) numerator/denominator; };}
```

*Specifica*: Elencare nella intestazione del metodo i tipi delle eccezioni sollevate (*throws*)

*Lancio*: Creazione di un oggetto di tipo eccezione (es: `ArithmeticException` è una sottoclasse della `RunTimeException`) + lancio della eccezione (*throws*)

Se si utilizzano librerie predefinite, controllare sulle API i tipi delle eccezioni sollevate dai metodi invocati

# GESTIONE DELLE ECCEZIONI: INTERCETTAZIONE+GESTIONE

Il programma P che invoca la *ArithmeticException* può:

decidere di non gestire la eventuale eccezione sollevata ma di *propagarla* ai livelli superiori  $\Rightarrow$  la eccezione deve essere dichiarata nella intestazione di P

*intercettare e gestire* la eccezione

intercettazione  $\Rightarrow$  *try + catch*

```
public static void main (String args[ ])
```

```
{ int x,y;
```

```
double ris; boolean ecc=false;
```

```
try
```

```
{ x= Integer.parseInt (args[0]);
```

```
y= Integer.parseInt (args[1]);
```

```
ris= DivideByZeroTest.quotient (x,y);
```

```
System.out.println (ris);
```

```
} catch (ArithmeticException e) {System.out.println (e); ecc=true;}
```

```
if (!ecc) {System.out.println ("non si e' verificata l'eccezione");}
```

```
}
```



# GESTIONE DELLE ECCEZIONI: INTERCETTAZIONE+GESTIONE

*Blocco try-catch* = blocco di codice racchiuso in parentesi graffe tra try e catch

*Gestore delle eccezioni* = blocco di codice racchiuso tra le parentesi graffe che seguono la clausola catch.

Semantica *try + catch*:

- Esecuzione delle istruzioni all'interno del blocco *try-catch*;
- Se le istruzioni ed i metodi eseguiti non sollevano eccezioni, si prosegue con la prima istruzione dopo il blocco try+catch;
- Se viene sollevata una eccezione, si interrompe l'esecuzione del blocco try+catch e si passa il controllo al gestore delle istruzioni. Dopo l'esecuzione del gestore, si prosegue con la prima istruzione dopo il blocco try-catch

# GESTIONE DELLE ECCEZIONI: INTERCETTAZIONE+GESTIONE

```
public static void main (String args[ ])  
  
{ int x,y;  
  double ris; boolean ecc=false;  
  try  
{ x= Integer.parseInt (args[0]);  
  y= Integer.parseInt (args[1]);  
  ris= DivideByZeroTest.quotient (x,y);  
  System.out.println (ris);  
} catch (ArithmeticException e) {System.out.println (e); ecc=true;}  
if (!ecc) {System.out.println (e);}  
}
```

# GESTIONE DELLE ECCEZIONI: ALCUNI METODI UTILI

Su un oggetto di tipo eccezione è possibile invocare tutti i metodi di base che possono essere invocati sul tipo base *Throwable*

*void printStackTrace()* visualizza lo stack delle chiamate, cioè la sequenza dei metodi invocati fino al punto in cui si è verificata l'eccezione:

Esempio1: Se inserisco nell'esempio precedente `e.printStackTrace()`

```
java.lang.ArithmeticException: hai diviso per 0  
  at DivideByZeroTest.quotient(DivideByZeroTest.java:4)  
  at provaDivide.main(provaDivide.java:11)
```

*String e.getMessage()* restituisce il messaggio che descrive la eccezione

# GESTIONE DELLE ECCEZIONI: USO DELLA PRINTSTACKTRACE()

```
public class Esempio {  
    public static void main (String args[])  
        { try { method1 ( );  
            } catch (Exception e) {e.printStackTrace( );}  
        };  
    public static void method1( ) throws Exception  
        { method2 ( );  
    }  
    public static void method2( ) throws Exception  
        { method3 ( );  
    }  
    public static void method3( ) throws Exception  
        { throw new Exception("Ho sollevato una eccezione");}  
}
```

# MULTI THREADING:MOTIVAZIONI

possibilità di definire più *flussi di esecuzione* all'interno dello stesso programma;

se il programma viene eseguito su un multiprocessor  $\Rightarrow$  i threads vengono eseguiti simultaneamente  $\Rightarrow$  miglioramento delle *prestazioni del programma*;

se il programma viene eseguito su un uniprocessor  $\Rightarrow$  i threads vengono eseguiti in *time-sharing* sullo stesso processore  $\Rightarrow$  l'esecuzione dei diversi flussi di esecuzione avanza concorrentemente

Non una panacea per aumentare le prestazioni, ma uno strumento per scrivere software *robusto e responsivo*, ad esempio

consideriamo *un browser* : mentre viene caricata una pagina, mostra una animazione. Inoltre mentre carico la pagina posso premere il bottone di stop ed interrompere il caricamento. Le diverse attività possono essere associati a threads diversi.

realizzazione di applicazioni client/server: ogni diversa richiesta di connessione dei clients può essere elaborata in un thread diverso (possibilmente su processori diversi)

# MULTI THREADING IN JAVA

Come si possono creare i Threads?

*Primo metodo*

- derivare una classe dalla classe *Thread* ed effettuare un *overriding* del metodo *run()*
- istanziare un oggetto della sottoclasse creata ed invocare il metodo *start()* sull'oggetto istanziato

```
public class FooThread extends Thread {  
    public void run ( ) {  
        for (int i=0;i<100;i++) {  
            System.out.println ("i="+i++); }  
    }  
    public static void main (String [ ] args) {  
        FooThread myFooThread = new FooThread ( );  
        myFooThread.start();  
    }  
}
```

# MULTI THREADING IN JAVA

## *Secondo Metodo*

- creare una classe che implementa l'interfaccia *Runnable* () ed implementare il metodo *run()* definito in questa interfaccia
- istanziare un oggetto di questa classe
- rendere questo oggetto un thread

```
public class FooRunnable implements Runnable {  
    public void run () {  
        for (int i=0;i<100;i++) {  
            System.out.println ("i="+i++); } }  
  
    public static void main (String [ ] args) {  
        FooRunnable myFooRunnable = new FooRunnable ( );  
        Thread fooThread = new Thread (myFooRunnable);  
        fooThread.start();  
    }  
}
```

# MULTI THREADING IN JAVA

Come scegliere tra i due metodi?

In JAVA una classe può estendere una sola altra classe (*eredità singola*)

⇒

il primo metodo può essere utilizzato *solo* quando la classe i cui oggetti devono essere eseguiti come threads non deve estendere altre classi

Secondo metodo: gli oggetti Thread possono appartenere ad una classe classe che estende una classe arbitraria

Esempio: Gestione degli eventi (es: movimento mouse, tastiera...) la classe che gestisce l'evento

- deve estendere una classe predefinita JAVA
- è naturale eseguire il gestore come un thread separato



# MULTI THREADING IN JAVA

- Stati di un thread: *running, blocked, finished*
- L'esecuzione di un thread termina quando il metodo `run()` ha completato l'esecuzione.
- L'esecuzione di un thread può essere bloccata/sbloccata a causa di diversi eventi
  - Esecuzione di una operazione di I/O
  - metodi `wait()`, `sleep()`, `yeld()`, `notify()`, `notifyall()`, `interrupt()`.....
- Metodo `interrupt()`, consente di risvegliare un thread

# MULTI THREADING IN JAVA: SINCRONIZZAZIONE TRA THREADS

```
import java.util.*;
class Buffer
{private Vector objects;

public Buffer (){
    objects = new Vector();}

public Object extract() throws InterruptedException {
    synchronized (objects) {
        while (objects.isEmpty()){
            objects.wait();
        }
        Object object = objects.firstElement();
        objects.removeElementAt(0);
        return object;
    }}

public void insert (Object object){
    synchronized (objects){
        objects.addElement(object);
        objects.notify();
    }}}
```

# MULTI THREADING IN JAVA: SINCRONIZZAZIONE TRA THREADS

```
public class Consumer1 extends Thread

{
    private Buffer buffer;

    public Consumer1(Buffer b){
        this.buffer=b;
    }

    public void run()
    {
        try{
            while (true)
                { Object object = buffer.extract();

                System.out.println(object);
            } } catch (InterruptedException ignored){ }
        }
    }
}
```

# MULTI THREADING IN JAVA: SINCRONIZZAZIONE TRA THREADS

```
public static void main (String args[ ]) throws InterruptedException{
```

```
    Buffer buffer = new Buffer();
```

```
    Consumer1 consumer = new Consumer1(buff);
```

```
    consumer.start( );
```

```
    for (int i=0;i<20;i++)
```

```
        {int casual= (int)(Math.random()*5000);
```

```
        buffer.insert(casual);
```

```
        }
```

```
    }
```

```
}
```

# ESERCIZIO

Il laboratorio di Informatica del Polo Marconi è utilizzato da tre tipi di utenti, studenti, tesisti e professori ed ogni utente deve fare una richiesta al tutor per accedere al laboratorio. I computers del laboratorio sono numerati da 1 a 20. Le richieste di accesso sono diverse a seconda del tipo dell'utente:

- a) i professori accedono in modo esclusivo a tutto il laboratorio, poichè hanno necessità di utilizzare tutti i computers per effettuare prove in rete.
- b) i tesisti richiedono l'uso esclusivo di un solo computer, identificato dall'indice *i*, poichè su quel computer è installato un particolare software necessario per lo sviluppo della tesi.
- c) gli studenti richiedono l'uso esclusivo di un qualsiasi computer.

I professori hanno priorità su tutti nell'accesso al laboratorio, i tesisti hanno priorità sugli studenti.

Scrivere un programma JAVA che simuli il comportamento degli utenti e del tutor. Il programma riceve in ingresso il numero di studenti, tesisti e professori che utilizzano il laboratorio ed attiva un thread per ogni utente. Ogni utente accede *k* volte al laboratorio, con *k* generato casualmente. Simulare l'intervallo di tempo che intercorre tra un accesso ed il successivo e l'intervallo di permanenza in laboratorio mediante il metodo `sleep`. Il tutor deve coordinare gli accessi al laboratorio. Il programma deve terminare quando tutti gli utenti hanno completato i loro accessi al laboratorio.

# PROGRAMMAZIONE DI RETE: INTRODUZIONE

*Sistema Distribuito* = insieme di hosts (computers) indipendenti , collegati mediante una Rete, e che possono collaborare per l'esecuzione di un task.

*Distributed Computing (Network Computing)* elaborazione effettuata da un sistema distribuito

*Parallel Computing* utilizzazione simultanea di più processori per eseguire un singolo Programma con alti requisiti computazionali. Scopo: aumento delle prestazioni del (compute-intensive applications, previsioni del tempo, grafica, astronomia,...)

# PROGRAMMAZIONE DI RETE: INTRODUZIONE

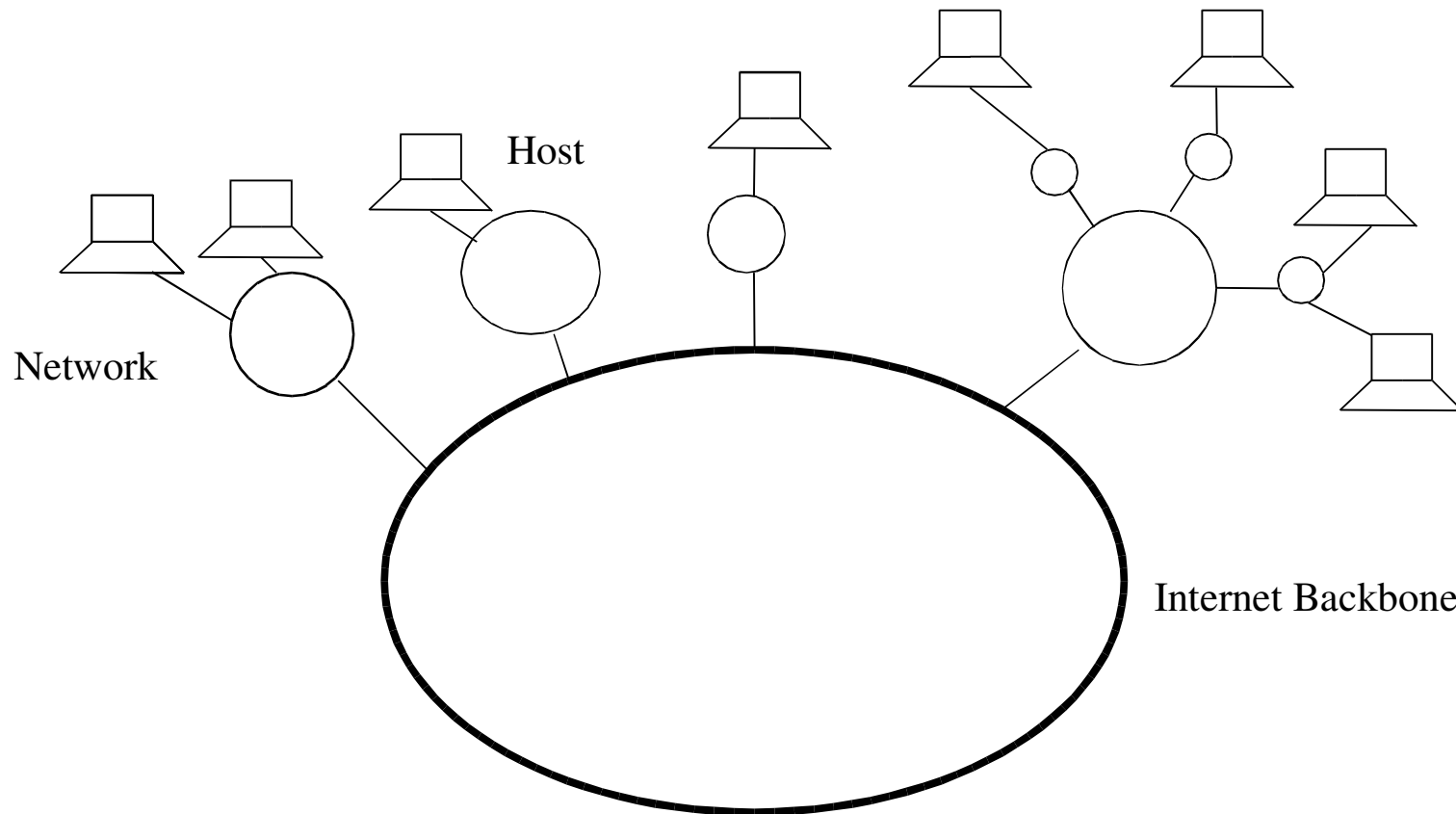
*Network Computing*: per poter collaborare alla realizzazione di un task, due o più *processi* in esecuzione su *hosts diversi*, distribuiti sulla rete, devono *comunicare*, ovvero scambiarsi informazioni

Comunicazione = Utilizza protocolli (= insieme di regole che i partners della comunicazione devono seguire per poter comunicare) ben definiti

Alcuni protocolli utilizzati in INTERNET:

- TCP (Transmission Control Protocol) un protocollo connection-oriented
- UDP (User Datagram Protocol) un protocollo connectionless

# PROGRAMMAZIONE DI RETE: MECCANISMI DI IDENTIFICAZIONE DEGLI HOSTS





# PROGRAMMAZIONE DI RETE: MECCANISMI DI IDENTIFICAZIONE DEGLI HOSTS

Identificare un processo con cui si vuole comunicare

⇒

Occorre identificare

- *la rete* all'interno della quale si trova l'host su cui e' in esecuzione il processo
- *l'host* all'interno della rete
- *il processo* in esecuzione sull'host

Identificazione dell'host = definita dal protocollo IP (Internet Protocol)

Facciamo riferimento ad IPv4 (IP versione 4). IPv6 (versione 6) si basa sugli stessi principi

Identificazione del processo = utilizza il concetto di *porta*

Porta = Intero da 0 a 65535

# PROGRAMMAZIONE DI RETE: MECCANISMI DI IDENTIFICAZIONE DEGLI HOSTS

In IPv4, ogni host sulla rete è identificato da una stringa di 32 bits (4 bytes o 4 *ottetti*)

Indirizzi IP assegnati dalla Internet Assigned Number Authority (IANA)

*Dotted Decimal Notation =*

