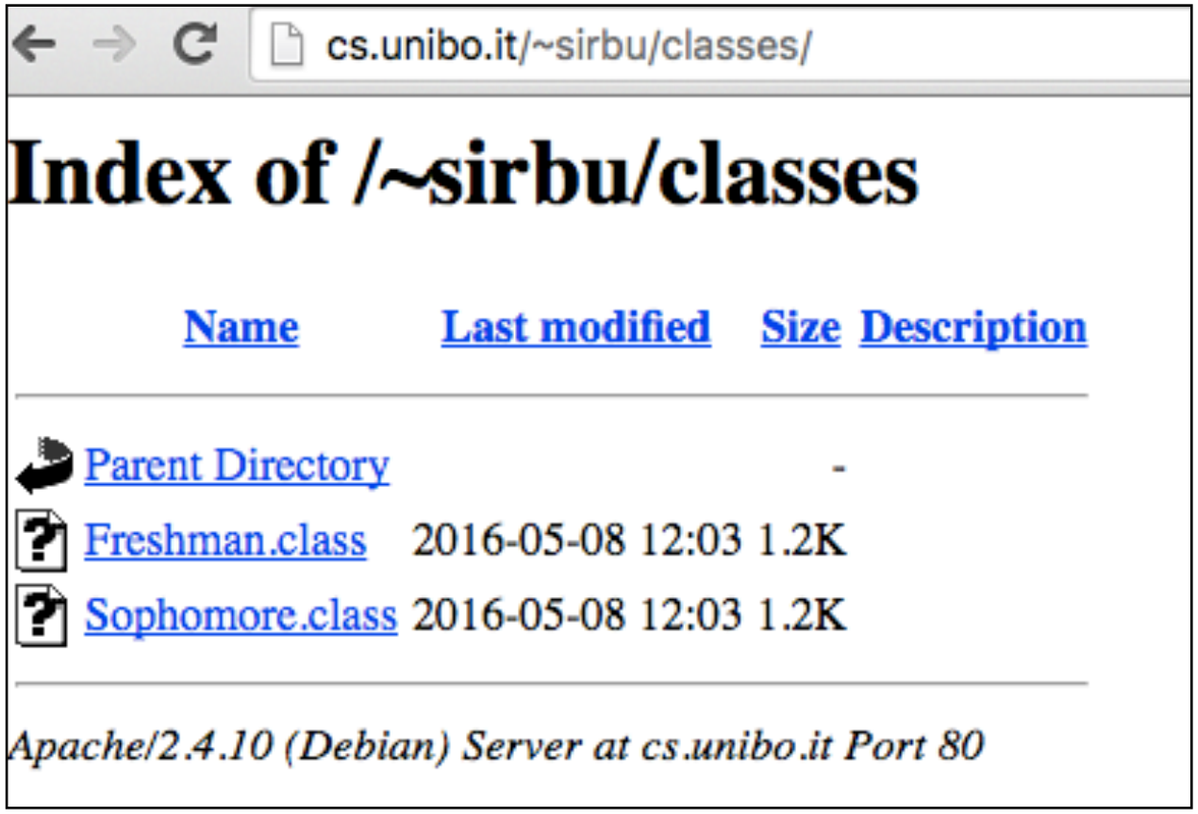
UNIVERSITÀ DI PISA

Programmazione di reti
Corso B

17 Maggio 2016
Lezione 11

Dynamic class loading - polimorfismo

- Mettere a disposizione le classi addizionali.



Dynamic class loading - polimorfismo

- Informare il cliente da dove può scaricare le classi - due possibilità:

    1. Lanciare il cliente indicando il *codebase* - come prima per lo *stub*

    2. Includere il *codebase* nel *server* -

        - durante la serializzazione gli oggetti vengono annotati con l'URL del *codebase* del *server*

        - il cliente scarica automaticamente le classi dall'URL ricevuto con gli oggetti

        - il cliente deve avere l'opzione `java.rmi.server.useCodebaseOnly=false` (altrimenti usa solo il codebase locale e non quello inviato dal *server*)

- La seconda opzione offre più flessibilità: la *codebase* può cambiare sul *server* senza dover informare i clienti

Dynamic class loading - polimorfismo

- I possibilità
  - lancio *server* senza parametri
  - lancio cliente con

```
-Djava.security.policy=permission
-Djava.rmi.server.codebase= http://cs.unibo.it/
~sirbu/classes/
```

```
Proxy[StudentManager,RemoteObjectInvocationHandler[UnicastRef [liveRef: [endpoint:
[192.168.1.73:63818](remote),objID:[-98ef99:1548fdb609a:-7fff, 805403467835634289 2]]]]]
Received students:
Freshman - Robert Brown living at 12 Dawson street. Student id 0. Current grade -1.0
Sophomore - Ann Brown living at 132 Buffallo street. Student id 3. Current grade -1.0
Received students:
Freshman - Robert Brown living at 12 Dawson street. Student id 0. Current grade 30.0
Sophomore - Ann Brown living at 132 Buffallo street. Student id 3. Current grade -1.0
```

Dynamic class loading - polimorfismo

- II possibilità
  - lancio *server* con `-Djava.rmi.server.codebase=http://cs.unibo.it/~sirbu/classes/`
    - lancio cliente con `-Djava.security.policy=permission -D java.rmi.server.useCodebaseOnly=false`

```
Proxy[StudentManager,RemoteObjectInvocationHandler[UnicastRef [liveRef: [endpoint:
[192.168.1.73:63818](remote),objID:[-98ef99:1548fdb609a:-7fff, 805403467835634289 2]]]]]
Received students:
Freshman - Robert Brown living at 12 Dawson street. Student id 0. Current grade -1.0
Sophomore - Ann Brown living at 132 Buffallo street. Student id 3. Current grade -1.0
Received students:
Freshman - Robert Brown living at 12 Dawson street. Student id 0. Current grade 30.0
Sophomore - Ann Brown living at 132 Buffallo street. Student id 3. Current grade -1.0
```

Contenuti

- Descrizione progetto
- `ThreadLocal`, *daemon thread*, `PipeInputStream/PipeOutputStream`
- Esercitazione: *server* per **MiniChatRoom**

Progetto: *SimpleSocial*

- Creare un'applicazione di *social networking*
- 2 componenti:
  - `SocialServer` - riceve richieste dai clienti, memorizza gli utenti, gli utenti online, la rete di amici, la rete di *follower*, inoltra messaggi agli utenti
  - `SocialClient` - gestisce input dal cliente e comunica col *server*

Registrazione, *login* e *logout*

- Quando viene avviato `SocialClient`, l'utente può fare o *login*, o la registrazione.
- Per la registrazione, l'utente invia ***username* e *password*** e riceve la conferma, su una connessione TCP.
- Per il *login*, l'utente invia ***username* e *password*,** e riceve un *token* di autenticazione dal server, su una connessione TCP. L'utente deve usare questo *token* per tutte le comunicazioni successive. Il *token* è valido per 24 ore. Se il *token* scade, `SocialClient` chiede all'utente di logarsi di nuovo. **Non si deve gestire la situazione in cui un utente fa il *login* due volte di seguito**.
- Per fare *logout*, l'utente invia la richiesta e il *server* conferma, su una connessione TCP.
- L'utente può **chiudere l'applicazione senza fare *logout*.** In questo caso, il ***logout* avviene automaticamente sul *server***. La prossima volta che l'applicazione viene avviata, si mostrano le due opzioni: *login* e register.
- Tutte le connessioni TCP vengono chiuse al completamento di ogni funzionalità.

Richiesta amicizia

- Un utente **u1** può richiedere di diventare amico con **u2**. `SocialClient(u1)` invia la richiesta al *server* su una connessione TCP.
- Se utente **u2** è online:
    - Il *server* invia la richiesta a `SocialClient(u2)` su una connessione TCP e risponde con una conferma al `SocialClient(u1)`. Questa è la conferma dell'inoltro della richiesta, non è la conferma dell'amicizia.
    - `SocialClient(u2)` memorizza la richiesta localmente.
- Se utente **u2** non è online, il *server* risponde con un messaggio di errore a `SocialClient(u1)`.
- Per verificare se **u2** è online, **non basta che il *server* guarda la sua lista di utenti online**, perché questa si aggiorna solo ogni tanto. Se il s*erver* riesce a inoltrare la richiesta a `SocialClient(u2)` vuol dire che **u2** è sicuramente online.
- La richiesta di amicizia non viene mostrata a **u2** subito, per facilitare l'uso di un'interfaccia testuale. **u2** deve richiedere lui stesso di vedere le richieste (questo comportamento può essere cambiato se si usa un'interfaccia grafica).

Conferma amicizia

- Utente **u2** sceglie di visualizzare le richieste di amicizia ricevute, `SocialClient(u2)` gli mostra le richieste memorizzate localmente.

- Utente **u2** sceglie un altro utente (diciamo **u1**) per cui confermare/negare l'amicizia. `SocialClient(u2)` invia la scelta dell'utente al *server*. Il *server* verifica se la richiesta di amicizia iniziale esiste nel sistema:
    - se esiste, modifica la rete di amici se necessario e risponde a **u2** con un messaggio di conferma. La richiesta di amicizia viene rimossa dalla lista sul *server*
    - se non esiste, risponde a **u2** con un messaggio di errore
- La richiesta viene rimossa anche dalla lista di `SocialClient(u2)`
- Se **u2** fa ***logout* senza guardare la lista di richieste,** la lista viene salvata da `SocialClient(u2)` sul disco e ricaricata quando **u2** richiede la visualizzazione.

Keep-alive

- Funzionalità che usa UDP e *multicast*
- Ogni 10 secondi il *server* invia un *multicast* ai clienti.
- I clienti devono rispondere al *server* con un **datagram (non multicast)**.
- I clienti che non rispondono entro 10 secondi, vengono considerati *offline*. Il *server* stampa il numero di clienti online.
- Lo stato *online/offline* viene aggiornato anche con ogni messaggio TCP o RMI ricevuto dai clienti.
- Domanda possibile: non basta che i clienti inviano un *datagram* ogni 10 secondi? - sarebbe una possibilità (anche più efficiente) però vorremmo **usare il *multicast*** comunque.

Richiesta lista amici e ricerca utenti

- Lista amici: `SocialClient` invia richiesta al *server*, il *server* risponde con una lista di nomi di amici e il loro stato *(online/offline*)

- Ricerca utenti: `SocialClient` invia una stringa al *server*, il *server* risponde con una lista di nomi di utenti che contengono la stringa.

- Le liste possono essere vuote.

- Si usano connessioni TCP

Contenuti: pubblicazione e *follower*

- Funzionalità che usano RMI
- Un utente **u1** si può **dichiarare *follower*** di un altro utente **u2**, e deve **registrare al server una *callback*** RMI, con cui il server può contattare **u1** quando **u2** pubblica contenuti.
- Il server offre il servizio di **registrazione di *callback* usando RMI**. `SocialClient(u1)` registra la callback **subito dopo il *login***.
- Il server offre il servizio di **diventare follower usando RMI**. L'utente **u1** sceglie di diventare follower di **u2** e `SocialClient(u1)` invoca il metodo remoto disponibile sul server.
- Il server memorizza la *callback* e anche la **rete di *follower*** (**u1** è follower di **u2**).
- Quando **u2** genera un contenuto (testo), il *server* richiama le *callback* di tutti i suoi *follower* (incluso **u1**).
- La *callback* non fa altro che memorizzare i contenuti ricevuti in una lista locale su `SocialClient(u1)`
- Il contenuto non viene mostrato subito all'utente **u1**, per facilitare lavoro con interfaccia testuale (in caso di interfaccia grafica, i contenuti possono essere mostrati anche subito).
- Se **u1** non è online, i contenuti vengono **memorizzati sul server** e inviati al `SocialClient(u1)` dopo che si fa il *login*, usando la nuova *callback* registrata dall'utente.

Visualizzazione contenuti

- **Operazione locale** sul `SocialClient`

- Utente richiede di vedere i contenuti già ricevuti (e memorizzati usando il *callback*)

- `SocialClient` li stampa e li rimuove dalla lista di contenuti.

- Se l'utente fa *logout* o chiude l'applicazione **senza vedere i contenuti**, la lista viene **salvata sul disco** e ricaricata quando l'utente richiede la visualizzazione.

Interfaccia

- Testuale: di base
- Grafica:
  - Richieste di amicizia possono essere visualizzate subito quando arrivano (usando un'area della interfaccia grafica, o un *popup*).
  - Contenuti degli amici possono essere mostrati subito in un'area dell'interfaccia grafica.
  - L'interfaccia grafica non è la parte principale del progetto - deve solo facilitare l'interazione con le funzionalità di `SimpleSocial`

Sottomissione

- **Solo se superati esami: Architettura e Sistemi Operativi**

- File di codice e *input*, configurazione, etc.

- Relazione in formato PDF con informazioni su: *design*, classi, *thread*, strutture dati, sincronizzazione.

- 2 eseguibili: uno per cliente, uno per server.

- **Solo formato elettronico (per corso B)**.

Sottomissione

- Sul sito moodle entro il *deadline* (alle **23:55!!!!**)
- Ci sarà un *deadline* per ogni appello di esame (**6 giugno** e **4 luglio** per l'estate)
- Si può sottomettere il progetto per il *deadline* di un'appello e discuterlo in un appello successivo, o fare lo scritto in un appello successivo. L'unico vincolo è: se si vuol fare lo **scritto nell'appello n**, si deve superare la discussione del **progetto almeno nell'appello n**.
- Non si deve fare l'iscrizione all'esame se si intende fare solo il progetto. **L'iscrizione all'esame si fa per lo scritto** (se si vuole andare allo **scritto** all'appello **n** si fa l'iscrizione per l'appello **n**, a prescindere del fatto che il progetto si discute nell'appello **n-i o n**)

Discussione

- La sottomissione del progetto riceverà un voto: 0 se non ammesso alla discussione, 1 se ammesso.

- Se ammesso, riceverete un *link* a un poll *Doodle* dove potete prendere appuntamento per la discussione.

- Durante la discussione si presenta il progetto (dimostrazione del funzionamento più risposta a delle domande sul codice) e si risponde ad altre domande relative agli argomenti studiati.

- Per ogni appello ci sara un giorno per la discussione (o 2 giorni se necessario) . Per questo estate, i giorni sono **9/10 giugno** per il primo appello, **7/8 luglio** per il secondo appello.

- **Portare student ID.**

Discussione - bibliografia

- Le *slide*, gli esempi e gli appunti
- Documentazione Java 8
- *Multithreading*:
  - *Java concurrency tutorial*: https://docs.oracle.com/javase/tutorial/essential/concurrency/index.html
  - Oaks & Wong, “Java Threads”, O’Reilly
  - Lewis & Berg, “Multithreaded programming with java technology”, Sun Microsystems
- *Socket*:
  - *Java networking tutorial*: https://docs.oracle.com/javase/tutorial/networking/index.html
  - Harold, “Java Network Programming”, O’Reilly
- RMI:
  - *Java RMI tutorial*: https://docs.oracle.com/javase/tutorial/rmi/
  - Architettura RMI : https://www.cis.upenn.edu/~bcpierce/courses/629/jdkdocs/guide/rmi/spec/rmi-arch.doc.html

Argomenti vari

Come usare le costanti

- 4 possibilità - dalla meno alla più flessibile (e consigliabile):

    1. non definite come variabili e usate ‘hard coded’ direttamente nel codice: ogni volta che dobbiamo cambiare il valore, dobbiamo cercare tutti gli utilizzi della costante e cambiarla. Molto poco flessibile, **non consigliata**!

    2. definite come variabili `final static` in una classe : ogni volta che vogliamo cambiare un valore, cambiamo solo la definizione. Però comunque dobbiamo ricompilare il codice - non suggerito per applicazioni che vengono poi distribuite a clienti

    3. ricevute come parametro alla riga di comando: valori possono essere cambiati senza compilare, però l’usabilità per clienti non-informatici è ridotta, anche quando si devono usare i parametri di *default*.

    4. definite in un file di configurazione: valori possono essere cambiati senza compilare, avvio dell’applicazione con i parametri di *default* molto facile.

ThreadLocal

- Quando si lavora con i *thread*, vogliamo avere variabili non condivisi dai *thread* (locali ai *thread*).

- Un modo è usare attributi privati dei `Runnable`, con un `Runnable` diverso per ogni *thread* - non funziona se abbiamo bisogno di un attributo statico.

- Un'altro modo è usare `ThreadLocal` - si crea una copia della variabile in ogni *thread*- funziona anche con attributi statici - abbiamo un valore per ogni thread, non per l'intero programma.

ThreadLocal

- Per create un attributo *threadlocal*:

```
private static ThreadLocal myThreadLocal = new
ThreadLocal<String>();

myThreadLocal.set(“This value is ThreadLocal");
String threadLocalValue = myThreadLocal.get();
```

- Per inizializzare l’oggetto per ogni *thread*:

```
private static ThreadLocal myThreadLocal = new
ThreadLocal<String>() {
  @Override protected String initialValue() {
  return “An initial value";
  }
};
```

Daemon thread

- *Thread* che esegue nel *background*

- Un'applicazione può finire anche se i suoi *thread daemon* sono ancora attivi.

- Quindi: un'applicazione finisce quando tutti i suoi *thread* non-*daemon* finiscono.

- In Java: la classe `Thread` ha il metodo `void setDaemon(boolean)` - imposta il comportamento daemon.

```
Thread t = new Thread(new MyRunnable);
t.setDaemon(true);
t.start();
```

Piped input/output stream

- *pipe* = tubo
- Un *pipe* può essere usato per comunicazione tra *thread*:
    - Un *thread* **A** apre un `PipedInputStream` e un *thread* **B** apre un `PipedOutStream`.
        - Si crea una connessione tra i due *stream*.
        - **A** può leggere i contenuti che **B** scrive
- Alternativa per il problema produttore-consumatore - invece di usare sincronizzazione su un oggetto condiviso, o una `BlockingQueue`.
- Offre i vantaggi degli *stream* : scrivere vari tipi di dati, *read/write* bloccante.

```
public class Mother implements Runnable{

    private ObjectOutputStream out;
    public Mother(ObjectOutputStream out) {
        this.out=out;
    }
    @Override
    public void run() {
        try {
            for(int i=0;i<10;i++){
                String food="Food "+i;
                out.writeObject(food);
                System.out.println("Cooked "+food);
                Thread.sleep(ThreadLocalRandom.current().nextInt(1000));
            }
        } catch (IOException e) {
            e.printStackTrace();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

}
```

```
public class Child implements Runnable{

   private ObjectInputStream in;

   public Child(ObjectInputStream in) {
      this.in=in;
   }
   @Override
   public void run() {
      try {
         for(int i=0;i<10;i++){
            String food=(String) in.readObject();
            System.out.println("Ate "+food);
         }
      } catch (IOException e) {
         e.printStackTrace();
      }catch (ClassNotFoundException e) {
         e.printStackTrace();
      }
   }

}
```

```
public class PipeMain {
    public static void main(String[] args) {

      ExecutorService es= Executors.newFixedThreadPool(2);
      try (PipedInputStream in= new PipedInputStream();
           PipedOutputStream out= new PipedOutputStream();){
         in.connect(out);
         Mother mom= new Mother(new ObjectOutputStream(out));
         Child kid = new Child(new ObjectInputStream(in));
         es.submit(kid);
         es.submit(mom);
         es.shutdown();
         es.awaitTermination(10, TimeUnit.MINUTES);
      } catch (IOException e) {
         e.printStackTrace();
      } catch (InterruptedException e) {
         e.printStackTrace();
      }

    }
}
```

```
Cooked Food 0
Cooked Food 1
Ate Food 0
Ate Food 1
Cooked Food 2
Cooked Food 3
Cooked Food 4
Ate Food 2
Ate Food 3
Ate Food 4
Cooked Food 5
Cooked Food 6
Ate Food 5
Ate Food 6
Cooked Food 7
Cooked Food 8
Ate Food 7
Ate Food 8
Cooked Food 9
Ate Food 9
```

Esercitazione

- Server del `MiniChatRoom`
  - con `Socket`
  - con `SocketChannel` e NIO