Java

dai fondamenti alla rete

G. Prencipe
prencipe@di.unipi.it

Presentazione

- Lo scopo di questo corso è di descrivere
  - Java come linguaggio di programmazione a oggetti (OO)
  - Java e la sua interazione con la rete
- La durata complessiva è di 80 ore
  - Lezione ogni martedì e giovedì, 9--13
  - Circa 50 ore saranno dedicate a presentare Java come linguaggio OO
  - Le restanti 30 saranno dedicate a descrivere come Java interagisce con la rete e come permette di realizzare applicazioni Web

Testi di riferimento Java

- Thinking in Java (in inglese)
  - Bruce Eckel
  - http://www.BruceEckel.com
  - Terza edizione gratuita
- Concetti di Informatica e Fondamenti di Java 2 (in italiano)
  - Cay S. Horstmann, Apogeo
- Java -- Guida alla programmazione (in italiano)
  - James Cohoon e Jack Davidson, Mc Graw Hill

Testi di riferimento Java e reti

- Java network programming (in inglese)
  - Elliotte Rusty Harold, O'Reilly
- Java network programming and distributed computing (in inglese)
  - David Reilly e Michael Reilly, Addison Wesley
- An introduction to network programming with Java (in inglese)
  - Jan Graba, Addison Wesley

Ambiente di sviluppo

- Java SE 1.4.2
  - Per servlet, Java EE
- Eclipse 3.1
  - http://www.eclipse.org

Programma di massima

- Introduzione
  - Descrizione generale delle caratteristiche di Java
- Tutto è un oggetto
  - Classi, metodi e dati
- Controllo del flusso di controllo
  - Operatori Java
  - Controllo dell'esecuzione
    - if, while, for,….
- Inizializzazioni
  - Costruttori
- Polimorfismo

Programma di massima

- Array e collezioni
- Errori e gestione delle eccezioni
- Il sistema di IO di Java
- Multithreading
- Utilizzo della rete
  - Socket TCP e UDP
  - Applicazioni web
  - JDBC
- Servlet
- JSP

Java

introduzione

G. Prencipe
prencipe@di.unipi.it

Iniziamo….

- In questa prima lezione, presenteremo a grandi linee gli aspetti che coinvolgono la Programmazione Orientata agli Oggetti (OOP)….
- ….tenendo sempre un occhio a Java (ovviamente!!)

Introduzione agli oggetti

- Perchè la programmazione orientata agli oggetti (OOP) si è diffusa negli ultimi anni?
- Diversi vantaggi a vari livelli
  - Promette sviluppo e mantenimento del software più veloce e meno costoso
  - Il processo di progettazione diviene più semplice e chiaro
  - L'eleganza e la chiarezza del modello ad oggetti e la potenza delle librerie rende l'attività di programmazione più semplice

Ma non c'è rosa senza....

- Lo scotto da pagare è senza dubbio legato alla curva d'apprendimento
- Pensare in termini di oggetti è senza dubbio un drastico cambiamento rispetto all'approccio procedurale
- La progettazione a oggetti è senza ombra di dubbio diversa da quella procedurale

Quale linguaggio

- In passato, chi si avvicinava al mondo del OOP, si trovava principalmente a dover scegliere tra
  - Smalltalk (Xerox PARC)
    - Era necessario conoscere enormi librerie prima di poter scrivere qualcosa
  - C++
    - Totale assenza di librerie
    - Questa situazione poi è migliorata con l'avvento di librerie fornite da terze parti e l'introduzione della libreria C++ standard

Quale linguaggio

- È difficile progettare bene oggetti
  - In realtà è difficile progettare bene qualsiasi cosa!!
- Lo scopo è quello di avere pochi programmatori esperti che mettono a disposizione di altri oggetti da utilizzare
- Un linguaggio OOP di successo non deve solo offrire una sintassi e un compilatore, ma un intero ambiente di sviluppo
  - Libreria di oggetti ben progettati e di facile utilizzo

Quale linguaggio

- Quindi, lo scopo di un programmatore OOP è di utilizzare bene oggetti già esistenti per risolvere i propri problemi e scrivere le proprie applicazioni
- A questo punto, ci concentriamo a presentare la programmazione OO, evidenziandone gli aspetti positivi (ovviamente!!)

L'astrazione

- Tutti i linguaggi di programmazione forniscono astrazioni
- Si può sostenere che la complessità dei problemi che si possono risolvere è direttamente collegata al tipo e alla qualità delle astrazioni disponibili
- La domanda è: cosa si astrae?

L'astrazione

- Assembler è una astrazione della macchina
- Molti linguaggi "imperativi" costituivano astrazioni dell'assembler (Fortran, Basic, C)
  - Essi comunque richiedono di pensare in termini della struttura del computer piuttosto che in termini della struttura del problema da risolvere
  - Il programmatore deve stabilire la connessione tra la macchina e il problema
  - Lo sforzo richiesto per effettuare questa associazione produce programmi difficili da scrivere e da mantenere

L'astrazione

- L'alternativa al modellare la macchina è rappresentata dal modellare il problema da risolvere
- Linguaggi come LISP e ASP modellano il mondo in modo particolare
  - Tutto è una lista
- Il PROLOG modella tutti i problemi in catene di decisioni
- Ognuna di queste soluzioni funziona bene nell'ambito per cui sono state progettate

L'astrazione

- L'approccio OO fornisce al programmatore strumenti per rappresentare gli elementi presenti nello spazio del problema
- Questa rappresentazione è sufficientemente generale da non costringere il programmatore a nessun tipo di problema particolare
- Questi elementi e la loro rappresentazione nello spazio delle soluzioni sono gli “oggetti”

Oggetti

- OOP permette di descrivere il problema in termini del problema, piuttosto che in termini della soluzione
- Ogni oggetto ha uno stato e delle operazioni che può effettuare
- Analogia con gli oggetti del mondo reale: essi hanno caratteristiche e comportamenti

Oggetti

- Possiamo riassumere le principali caratteristiche dell'OOP come segue (descritte per Smalltalk, si cui Java si basa)
  1. Tutto è un oggetto
  2. Un programma è un insieme di oggetti che si dicono cosa fare scambiandosi messaggi
  3. Ogni oggetto è costruito a partire da altri oggetti
  4. Ogni oggetto ha un tipo
  5. Tutti gli oggetti di un particolare tipo possono ricevere gli stessi messaggi

1. Tutto è un oggetto

- Un oggetto può essere visto come una variabile “particolare”
- Memorizza dati
- Ma può anche effettuare operazioni, quando richiesto
- In teoria, ogni componente concettuale del problema da risolvere può essere rappresentato da un oggetto

2. Un programma è un insieme di oggetti

- Per inviare una richiesta a un oggetto, bisogna mandargli un messaggio
- Un messaggio a un oggetto può essere visto come una richiesta di invocare una funzione che appartiene a quel particolare oggetto

3. Ogni oggetto è composizione di altri oggetti

- In altri termini, è possibile creare un nuovo tipo di oggetto definendo un pacchetto che contiene oggetti già esistenti
- Quindi, è possibile modellare concetti complessi in un programma, nascondendo le complessità dietro la semplicità degli oggetti

4. Ogni oggetto ha un tipo

- Ogni oggetto è una “istanza” di una classe, dove classe è sinonimo di tipo
- La caratteristica che distingue maggiormente una classe è “quali messaggi posso inviare ad essa?”

Oggetti

- Aristotele è stato forse il primo a parlare del concetto di tipo
- Egli parlava della “classe dei pesci e della classe degli uccelli”
- Il concetto che tutti gli oggetti, sebbene unici, sono anche parte di un insieme di oggetti che hanno caratteristiche e comportamenti unici è stato usato nel primo linguaggio OO, Simula-67
  - Introdusse la parola chiave **class**

Simula-67

- Simula è stato definito per sviluppare “simulazioni” come quella del classico problema del banchiere
- Ci sono un po' di addetti allo sportello, clienti, conti correnti, transazioni
- I membri (elementi) di ogni classe hanno particolari caratteristiche
  - Clienti (nome, numero conto)
  - Conto (saldo)
- Ognuno di essi può essere rappresentato da una entità unica: l'oggetto, che appartiene a una particolare classe che ne definisce le caratteristiche e i comportamenti

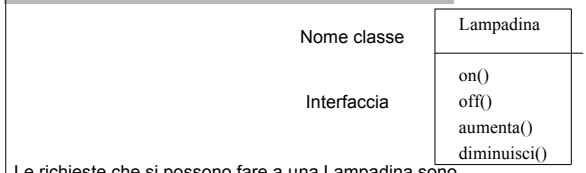
Oggetti

- Quando una classe (leggi anche: tipo) è stata stabilita, possono essere creati quanti oggetti si desiderano, e manipolare questi oggetti come gli elementi che esistono nel problema da risolvere
- Ma come ottenere un oggetto utile? Cioè che faccia quello che occorre per risolvere il problema?

Interfaccia

- Deve esistere un modo per fare richieste agli oggetti, in modo che essi facciano quello che vogliamo
- Le richieste che possono essere fatte a un oggetto sono definite dalla sua “interfaccia”

Esempio: la lampadina



Le richieste che si possono fare a una Lampadina sono di accendere, spegnere, aumentare e diminuire l'intensità

Per ottenere una Lampadina, si crea una "maniglia" per Lampadina, dichiarando un nome, e poi creando un oggetto di quel tipo con **new**

```
Lampadina lt = new Lampadina();
lt.on();    /*Invia un messaggio alla Lampadina
```

L'implementazione nascosta

- Introduciamo ora due attori importanti nella programmazione OO
  1. Creatori di classi
     - Coloro che creano nuovi tipi
  2. Programmatori clienti
     - Coloro che utilizzano nelle loro applicazioni le classi create

L'implementazione nascosta

- Lo scopo dei clienti è quello di collezionare classi da utilizzare nelle proprie applicazioni
- Lo scopo dei creatori di classi è di fornire classi che mostrano solo quello che è necessario ai programmatori
  - Tutto il resto è nascosto
  - Perché??

L'implementazione nascosta

- Se nascosto, allora il programmatore non lo vede e non lo può usare....
- ....e quindi il creatore di classi può modificare la porzione nascosta senza preoccuparsi dell'impatto che questo ha su chi usa quella classe!!
- In altre parole, le parti nascoste costituiscono il modo con cui è implementata l'interfaccia di un certo oggetto

L'implementazione

- L'interfaccia stabilisce **cosa** può essere richiesto a un certo oggetto
- Questo corrisponde a porzioni di codice
- Questo codice, insieme all'implementazione nascosta, costituisce l'implementazione

Controllo dell'accesso

- Risulta chiaro che non tutto è accessibile ai clienti
- Ci sono due motivi fondamentali per controllare l'accesso
  - Evitare che i clienti delle classi possano accedere alle implementazioni (cosa che non è necessaria per "usare" gli oggetti)
  - Permettere ai gestori delle librerie di modificare i funzionamenti interni senza preoccuparsi dell'effetto sui programmatori
    - Es.: decidere di ottimizzare (a posteriori) una certa implem.
    - Se interfaccia e implementazione sono chiaramente separate, tutto questo è più facile

Controllo dell'accesso

- Java utilizza tre parole chiave per delimitare gli accessi
  - **public**
    - Accessibile a tutti
  - **private**
    - Nessuno può accedere tranne il creatore e le altre funzioni definite per quel tipo
  - **protected**
    - Come private, con l'eccezione che sottoclassi possono accedere a membri che sono **protected**, ma non **private**

Riutilizzo del codice

- Dopo aver creato (e testato) una classe, essa dovrebbe rappresentare una utile unità di codice
- Purtroppo il riutilizzo del codice non è sempre facile da ottenere
- Il modo più semplice per riutilizzare una classe è quello di utilizzare un oggetto di quella classe direttamente (*oggetto membro*)
  - È possibile mettere un oggetto di quella classe dentro una nuova classe!!

Composizione

- Una nuova classe può essere costituita da un qualsiasi numero e tipo di altri oggetti
  - Dipende dalle funzionalità che vogliamo assegnare alla nuova classe
- Questo concetto è detto composizione
  - Si compongono classi a partire da altre classi

Ereditarietà

- Il concetto di oggetto è molto utile e pratico, dato che consente di rappresentare la realtà seguendo *concetti*
- Sarebbe comunque uno spreco se, una volta definito un tipo di dato (classe), dobbiamo rifare tutto da capo per definirne un altro molto simile
- In realtà, la OOP fornisce un meccanismo per non dover fare tutto da capo se vogliamo semplicemente aggiungere o modificare caratteristiche a un dato tipo: l'ereditarietà

Ereditarietà

- La classe originale viene detta *superclasse* o *padre*
- La classe modificata viene detta *sottoclasse* o *figlio*
- Se la superclasse viene modificata , i cambiamenti si riflettono sul figlio
- In Java, l'ereditarietà si ottiene con la parola chiave **extends** (da utilizzare quando si definisce una nuova classe)

Ereditarietà

- Nella sottoclasse, viene duplicata l'interfaccia della superclasse
  - I messaggi inviati alla superclasse possono essere inviati anche alla sottoclasse
  - Dato che il tipo di una classe è riconoscibile dai messaggi che possiamo inviare ad essa, la sottoclasse è dello stesso tipo della superclasse
  - Questa equivalenza dei tipi è cruciale nella OOP

Ereditarietà

- L'ereditarietà è chiaramente utile se differenziamo padre e figlio
- Possiamo fare questo
  - Aggiungendo nuove funzionalità al figlio
  - Cambiare le funzionalità del padre: overriding

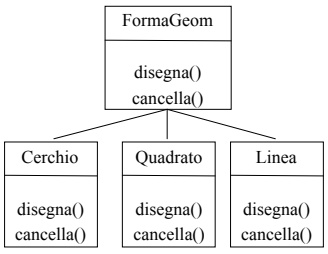
Overriding

- L'interfaccia non cambia (tra padre e figlio), cambia l'implementazione
- In pratica, il tipo derivato è lo stesso della classe padre
- Come risultato, un oggetto della classe derivata può essere tranquillamente sostituito con uno della classe padre
- Questo tipo di relazione viene detta di tipo **è un**
  - Il cerchio **è una** forma geometrica

Overriding

- Nel caso in cui aggiungiamo funzionalità al figlio, abbiamo una relazione **è come**
  - Il climatizzatore **è come un** condizionatore

Estensibilità



- Un oggetto della classe derivata è trattato come un oggetto di quella originale
- Se un nuovo tipo è introdotto (Triangolo) con ereditarietà, il codice funzionerà anche per la nuova FormaGeom introdotta
- Il programma è *estensibile*

Polimorfismo

- La seguente funzione invoca (invia messaggi a un oggetto) funzionalità di FormaGeom

```
void doStuff(FormaGeom s){
    s.cancella();
    ....
    s.disegna();
}
```

- Questa funzione comunica con ogni oggetto FormGeom. Quindi può essere invocata indipendentemente su un Cerchio, Quadrato, o Linea

Polimorfismo

```
Cerchio c = new Cerchio();
Quadrato q = new Quadrato();
doStuff(c);
doStuff(q);
```

- doStuff può essere invocata su oggetti di tipo diverso; l'importante è che siano di tipo FormaGeom
  - *Polimorfismo*
- Una entità di tipo Cerchio viene passata a una funzione che si aspetta una entità di tipo FormaGeom
- Dato che un Cerchio **è una** FormaGeom, tutto va bene
  - Upcasting (trattare un tipo derivato come se fosse quello originale)

Binding dinamico

- Quindi, un messaggio **disegna()** o **cancella()** viene inviato a una FormaGeom, e quello che poi accade dipende dal tipo a cui la FormaGeom è connessa (se Cerchio o Linea o....)
- A tempo di compilazione, però, il compilatore non sa, quando compila **doStuff()**, su cosa questa funzione sarà invocata
- Come funziona il tutto?

Binding dinamico

- Quando si invia un messaggio a un oggetto senza sapere esattamente di che tipo è, abbiamo polimorfismo
- Il processo utilizzato in OOP per implementarlo è detto **binding dinamico**
- Alcuni linguaggi necessitano di parole chiave per abilitarlo
  - **virtual** in C++
- In Java è automatico
  - Quando inviamo un messaggio a un oggetto, esso farà la cosa giusta, anche in presenza di upcasting

Classi astratte

- Spesso durante la progettazione si ha bisogno che una superclasse presenti solo un'interfaccia alle sue sottoclassi
- In altre parole, non vogliamo effettivamente che vengano creati oggetti della superclasse
- Questo si ottiene definendo la classe **abstract**
- Se si prova a creare un oggetto di una classe astratta, si ottiene un errore in compilazione

Metodi astratti

- Anche una funzionalità (metodo) può essere definito astratto
  - È un metodo che non è stato ancora implementato
- Un metodo astratto può essere creato solo in una classe astratta
- Quando la classe viene ereditata, il metodo deve essere implementato
  - Altrimenti tutta la sottoclasse diviene astratta
  - Serve per inserire una funzionalità in una interfaccia senza doverne dare immediatamente l'implementazione

Durata oggetti

- Tipicamente, tipi di dato astratti, ereditarietà e polimorfismo sono gli ingredienti principali nell'OOP
- Un altro fattore importante riguarda il modo con cui gli oggetti sono creati e distrutti
- Esistono diverse filosofie di pensiero

Creazione -- pila

- C++ approccia il problema sostenendo che l'efficienza è fondamentale
  - Lascia al programmatore una scelta
  - Per massimizzare la velocità d'esecuzione, è possibile determinare la durata degli oggetti mentre si scrive il programma, mettendo gli oggetti su una pila o in un'area di memoria statica
- Viene sacrificata la flessibilità dato che bisogna conoscere a priori il numero, la durata e il tipo degli oggetti da utilizzare

Creazione oggetti -- heap

- Il secondo approccio è di creare gli oggetti dinamicamente, utilizzando uno heap
- In questo approccio, non si conosce il numero e la durata degli oggetti fino al tempo d'esecuzione
- Se si ha bisogno di un nuovo oggetto, lo si mette sull'heap quando serve

Creazione oggetti -- heap

- Dato che lo spazio per memorizzare oggetti è gestito dinamicamente, il tempo necessario per allocare spazio sullo heap e maggiore di quello necessario per la pila
  - Creare spazio sulla pila tipicamente coinvolge una sola istruzione assembler per muovere il puntatore alla pila
- Questo approccio si basa sull'assunzione che gli oggetti tendono ad essere complessi, quindi il tempo impiegato per allocare lo spazio e poi rilasciarlo non ha un grande impatto sulla creazione di un oggetto
- Il C++ adotta anche questo approccio

Durata oggetti

- La conclusione sarebbe che, vista la flessibilità, è preferibile creare sempre gli oggetti sullo heap
- Qui sorge un nuovo problema: la durata degli oggetti
- Con un oggetto creato sullo stack o staticamente, il compilatore può facilmente determinare quanto esso dura, e distruggerlo quando non serve più
- Creandolo su un heap, il compilatore non ha alcuna informazione su di esso

Durata oggetti

- Ci sono due scelte
  - Decidere manualmente (a livello di programmazione) quando distruggere gli oggetti
  - L'ambiente può fornire uno strumento chiamato *garbage collector* (GC), che automaticamente scopre quando un oggetto non è più in uso e lo distrugge
  - Il GC chiaramente costa tempo
    - Per questo non è incluso nel C++
    - Ne esistono forniti da terze parti
  - Java ne ha uno (come Smalltalk)

Collezioni

- Spesso capita di non sapere quanti oggetti sono necessari per risolvere un certo problema
- Come si fa a sapere allora quanto spazio creare per questi oggetti?
- Una soluzione a questo problema prevede la creazione di un nuovo tipo di oggetto

Collezioni

- Questo nuovo tipo ha il compito di memorizzare altri oggetti: la *collezione* (**collection**)
- Chiaramente questo compito è tipico degli array
- La differenza è che questo nuovo tipo si auto espande a seconda della necessità
- Quindi non c'è bisogno di sapere in anticipo quanti oggetti conterrà una collezione
  - La si crea, e poi lei si preoccuperà del resto

Collezioni

- Un buon linguaggio OO contiene un certo numero di collezioni già definite
- Tutte le collezioni hanno un modo per inserire e prelevare elementi
  - push o add o simili, per inserire

Collezioni -- iteratore

- Estrarre non è così ovvio (estrarre solo un elemento non sempre ha senso)
- L'*iteratore* consente invece di selezionare gli elementi di una collezione e di restituirli
- Esso consente di esplorare la collezione
  - Nota: esso astrae da che tipo di collezione sta esplorando (il suo utilizzo è lo stesso sia che si tratti un vettore o di una lista)

Gerarchia delle classi

- In Java tutte le classi sono sottoclassi della stessa superclasse
- Questa superclasse si chiama **Object**
- I vantaggi di un approccio di questo tipo sono diversi

Unica superclasse

- Tutti gli oggetti hanno un'interfaccia in comune, quindi in ultima istanza sono tutti dello stesso tipo
- Tutti gli oggetti hanno sicuramente alcune funzionalità
  - C'è la garanzia che su tutti gli oggetti possono essere eseguite alcune operazioni di base
- L'implementazione del garbage collector è più semplice
  - Il supporto necessario può essere istallato nella superclasse, e il GC può mandare i messaggi appropriati a ogni oggetto nel sistema

Unica superclasse

- Questa filosofia non è adottata in C++
- Per ottimizzare l'efficienza
- Per mantenere una certa compatibilità con il C

Domanda

- Come fare ad ottenere una collezione che può contenere qualsiasi cosa?

Domanda

- Come fare ad ottenere una collezione che può contenere qualsiasi cosa?
- Una collezione che contiene **Object** può contenere qualsiasi cosa!!
- Per utilizzare questa collezione, semplicemente aggiungiamo oggetti ad essa
- Dato che la collezione contiene solo **Object**, quando vi aggiungiamo oggetti, otteniamo un loro *upcast* a **Object**, perdendo così la loro identità iniziale

Downcast

- Quando leggiamo gli oggetti inseriti nella collezione, dunque, otteniamo qualcosa che è **Object**, e non un l'oggetto con l'interfaccia che avevamo inserito
- Come facciamo a trasformarlo di nuovo in quello che avevamo?
- Si fa un *downcast*: si percorre la gerarchia degli oggetti verso il basso

Upcast e downcast

- Con l'upcasting sapevamo che il Cerchio è una FormaGeom
- Però non sappiamo necessariamente che una FormaGeom è un Cerchio (può essere Linea)
- Quindi il downcasting è più pericoloso da fare
  - Se si sbaglia si ottiene errore a tempo d'esecuzione (*eccezione*)

Templates

- La soluzione che evita problemi con il downcast prevede collezioni che conoscono il tipo degli oggetti che contengono
  - *Parameterized types*
- Questa soluzione è adottata in C++ con i *templates*
- Java li ha introdotti nella versione 1.5
  - *generics*

Eccezioni: gestire gli errori

- In Java la gestione degli errori è cablata direttamente nel linguaggio di programmazione
  - Avviene tramite le *eccezioni*
- Domanda: ma cosa sarà mai una eccezione in Java?

Eccezioni: gestire gli errori

- Una eccezione è (in piena filosofia Java) un oggetto che è "lanciato" (thrown) dal punto in cui è avvenuto l'errore
- Viene "catturato" (catch) da un *exception handler* specifico per l'eccezione generata
  - È un cammino parallelo che viene seguito nel caso in cui le cose vadano male

Eccezioni: gestire gli errori

- Dato che è un cammino parallelo, non interferisce con il normale codice
  - In questo modo il programmatore non è costantemente costretto a controllare per la presenza di errori
- Inoltre una eccezione non può essere ignorata, come un valore d'errore o un flag restituito da una funzione per indicare una condizione d'errore (come avviene in altri linguaggi)
  - Questo garantisce che bisognerà gestire l'eccezione in qualche punto

Eccezioni: gestire gli errori

- Forniscono un modo per recuperare da situazioni d'errore
  - Invece di terminare semplicemente, è generalmente possibile fare in modo di sistemare le cose per recuperare l'esecuzione corretta del programma
  - Programmi più robusti

Eccezioni: gestire gli errori

- Come già accennato, in Java la gestione delle eccezioni è obbligatoria
- Se il codice non viene scritto in maniera opportuna in modo da gestirle, si ottengono errori in compilazione
- Si nota comunque che la gestione delle eccezioni non è caratteristica dei soli linguaggi OO
  - ADA, alcune versioni di LISP

Multithreading

- Uno dei concetti fondamentali di programmazione è l'idea di gestire più di un *task* alla volta
- Molti problemi di programmazione infatti richiedono che il programma sia in grado di fermare l'attività corrente, gestire qualche altro problema, e poi ritornare all'attività iniziale

Multithreading

- Uno dei modi per ottenere questo è tramite *interrupt*
  - Soluzione costosa e poco portabile
- Gli interrupt sono sicuramente necessari per gestire una certa classe di compiti (soprattutto legati alla gestione del SO)
- Esiste comunque una grande classe di problemi per cui una soluzione è il partizionamento in sottoproblemi che vengono eseguiti separatamente
  - *Threads* e il concetto generale viene indicato con il termine di *multithreading*

Multithreading

- Con un singolo processore, i thread rappresentano solo un modo di allocare il tempo CPU
- Con più processori, i vari task possono essere effettivamente parallelizzati sulle varie CPU

Multithreading

- Avere multithreading a livello di linguaggio, consente al programmatore di non preoccuparsi se c'è una o più CPU
- Il programma viene logicamente diviso in threads, e del resto si occupa il SO

Multithreading

- Un problema con i thread è rappresentato dalla gestione delle risorse condivise
  - Più thread che accedono alla stessa risorsa
  - Es. più thread inviano informazioni a una stampante
- Per risolvere il problema, le risorse condivisibili vengono "bloccate" (lock)
  - Il thread blocca la risorsa, la usa e poi la rilascia

Multithreading

- In Java il multithreading è cablato nel linguaggio
- Il threading è gestito da una classe particolare
  - Un thread d'esecuzione è rappresentato semplicemente da un oggetto

Persistenza

- Tipicamente tutti gli oggetti creati durante l'esecuzione di un programma vengono distrutti quando il programma termina
- Ci sono varie situazioni in cui è comodo non distruggere tutti gli oggetti alla terminazione, ma avere un modo per conservarli
  - Quando riavviamo il programma essi esistono ancor con le stesse informazioni che contenevano al termine dell'ultima esecuzione
- Per ottenere questo effetto, si dichiara un oggetto *persistente*

Java e Internet

- Una delle ragioni principali per cui Java ha avuto un'enorme diffusione è il modo con cui interagisce con Internet e il Web
- Uno dei principali protocolli di comunicazione sul Web è quello client/server

Client/server

- L'idea è quella di avere una macchina server che fornisce le informazioni a un insieme di clienti che le richiedono
  - Le informazioni sono centralizzate sulla macchina server
  - Il compito del server è di distribuire le informazioni che detiene

Client/server

- Per *server* si intende l'insieme delle informazioni, il software che consente la distribuzione delle informazioni, e la macchina dove le informazioni e questo software risiedono
- Il *client* è costituito dal software che risiede sulla macchina remota che comunica con il server per ottenere le informazioni, le processa e le mostra
- I problemi maggiori derivano dal fatto che un singolo server cerca di soddisfare le richieste di più clienti (contemporaneamente)

Client/server

- Il Web in effetti è un enorme sistema client/server
- In realtà la situazione è anche più critica, dato che tipicamente su una stessa macchina coesistono sia il client che il server
- Inizialmente lo scenario era semplice
  - Si effettua un richiesta al server, che fornisce un file, che viene interpretato dalla macchina client

Client/server

- In breve, le necessità sono aumentate
  - Anche il cliente vuole inviare informazioni al server
  - In pratica, una comunicazione bidirezionale
- I browser web sono stati un passo in avanti
  - Un pezzo di informazione può essere visualizzato su qualsiasi tipo di computer senza alcun cambio
- Inizialmente comunque non erano molto interattivi
  - Semplice visualizzazione di pagine
- Diversi approcci per risolvere questo problema
  - Includere nel cliente la capacità di eseguire programmi sotto il controllo del browser: Client-side programming

CGI

- Inizialmente nei web browser l'interazione fra server e cliente era gestita solo dal server
  - Pagine html ricevute e interpretate dal cliente
  - Html di base fornisce meccanismi per l'inserimento di dati che possono essere inviati al server
  - Questo invio di dati passa attraverso la Common Gateway Interface (CGI)
    - Il testo inviato dice al CGI cosa fare
    - Tipicamente viene richiesta l'esecuzione di codice sul server
    - Questo codice può essere scritto in diversi linguaggi

CGI

- Il problema del CGI è il tempo di risposta
  - Dipende dalla quantità di dati inviati e dal carico del server e della rete
  - Es. validazione di dati da un input form
    - Preme *submit*
    - I dati sono inviati al server
    - Il server avvia il programma CGI
    - Scopre un errore e formatta una pagina html informando dell'errore
    - Invia la pagina al cliente, che torna indietro e prova di nuovo

Client-side programming

- La soluzione è rappresentata dal *client-side programming*
- I clienti tipicamente risiedono su macchine che con l'approccio descritto finora semplicemente attendono notizie dal server
- *Client-side programming* significa caricare il web browser con tutto il lavoro che può fare
  - Il risultato è un aumento di velocità e maggiore interattività

Plug-ins

- Uno dei passi verso il client-side programming è costituito dai *plug-ins*
- È un modo per aggiungere maggiori funzionalità al browser scaricando un pezzo di codice che si inserisce (plugs in) nel browser
  - Gli permette di eseguire qualche nuova attività
  - Permette a un programmatore esperto di definire un nuovo linguaggio e aggiungerlo al browser senza il permesso di chi ha definito il browser

Linguaggi di scripting

- L'introduzione dei plug-ins ha causato l'esplosione dei *linguaggi di scripting* (LS)
- Con un LS il codice del programma viene inserito direttamente nella pagina html
- Il plug-in che interpreta quel linguaggio viene automaticamente attivato mentre la pagina html viene visualizzata
- Sono tipicamente semplici
- Dato che sono nella pagina html, sono visibili

Linguaggi di scripting

- Tra i principali linguaggi di scripting possiamo citare
  - JavaScript
  - VBScript
  - Tcl/Tk
- Con l'utilizzo di un linguaggio di scripting, si riescono a risolvere circa l'80% dei problemi tipici che si incontrano nel client-side programming
  - Per gli altri bisgna ricorrere a qualcosa di più

Java

- Java consente client-side programming tramite le *applets*
  - Sono dei piccoli programmi che girano in un web browser
- Viene scaricata automaticamente come parte della pagina web
- Quando viene attivata esegue un programma sulla macchina cliente
  - In questo modo il server fornisce al client il software solo al momento in cui ne ha bisogno

Applets

- Dato che Java è un linguaggio completo, è tipicamente possibile far fare al cliente la maggior parte del lavoro
  - Es. non è necessario inviare moduli da compilare sulla rete, quando tutto può essere gestito sul cliente
  - Es. la computazione di grafici può essere fatta sul cliente, evitando che il server invii immagini di grandi dimensioni
- Un vantaggio sugli script è che una applet è in forma compilata, quindi non visibile
  - Anche se può essere decompilata senza grandi difficoltà

ActiveX

- ActiveX è sotto alcuni aspetti il "rivale" Microsoft di Java
- Inizialmente solo per Windows
  - Sviluppo per piattaforme generiche (da parte di consorzio indipendente)
- Filosofia diversa
  - Non costringe il programmatore a qualche linguaggio particolare
  - Se conosco C++ o VB, è possibile creare componenti ActiveX che consentono di effettuare client-side programming

Sicurezza

- Scaricare ed eseguire automaticamente programma può costituire una minaccia alla sicurezza
- Java è stato progettato per eseguire le applet in una "sandbox" di sicurezza
  - No accessi al disco o alla memoria al di fuori della sandbox
- Programmare in ActiveX è come programmare Windows
  - Si può fare ogni cosa
  - Un componente ActiveX può danneggiare i file sul computer

Sicurezza

- Una soluzione sono le *firme digitali* (*digital signatures*)
  - Il codice viene verificato per mostrare chi è l'autore
  - Si basa sull'idea che in un virus l'autore è anonimo
  - Non protegge da errori non voluti presenti nel codice

Sandbox in Java

- In Java si evita che i problemi avvengano, tramite la sandbox
- L'interprete Java presente nel web browser analizza la applet per scoprire istruzioni illegali mentra essa viene caricata
- In particolare le applet non possono scrivere o cancellare file (aspetti più pericolosi dei virus)

Signed applet

- Comunque, per permetter anche a particolari applet di scrivere su disco (es. costruzione database), sono state introdotte le *signed applet*
- Esse usano una chiave pubblica per verificare che una applet arrivi da dove dice di arrivare

Firme digitali

- Comunque, le firme digitali non tengono conto della velocità a cui si muove Internet
- Se viene scaricato un programma con un baco, quanto tempo passa prima che i danni causati vengano scoperti?
  - Giorni o anche settimane
  - E dopo questo tempo, come si scopre quale programma ha causato quali danni?

Server-side programming

- Anche il server può partecipare attivamente alle comunicazioni con il client
- Tipicamente gli viene chiesto semplicemente di inviare pagine html
- In scenari più complessi, le richieste possono essere processate tramite codice eseguito sul server (es. transazioni su database)
  - *Server-side programming*
  - In Java si ottiene tramite *servlets*

Analisi e progettazione

- La OOP è relativamente nuova e introduce un nuovo modo di affrontare i problemi e di pensare alle soluzioni
- Dato che in un approccio OOP tutto è un oggetto, bisogna chiedersi subito
  - Quali sono gli oggetti?
    - Come partizionare il problema nelle sue componenti?
  - Quali sono le interfacce?
    - Quali messaggi ho bisogno di inviare a ogni oggetto?

Analisi e progettazione

- Una parte delicata nella progettazione OOP è stabilire quali sono le classi di cui abbiamo bisogno e come esse devono interagire
- Uno strumento utile per descrivere questo è fornito dallo *Unified Modeling Language* (UML)
- L'importante è avere uno strumento comune che consenta di descrivere gli oggetti e le loro interfacce

Java
introduzione

fine