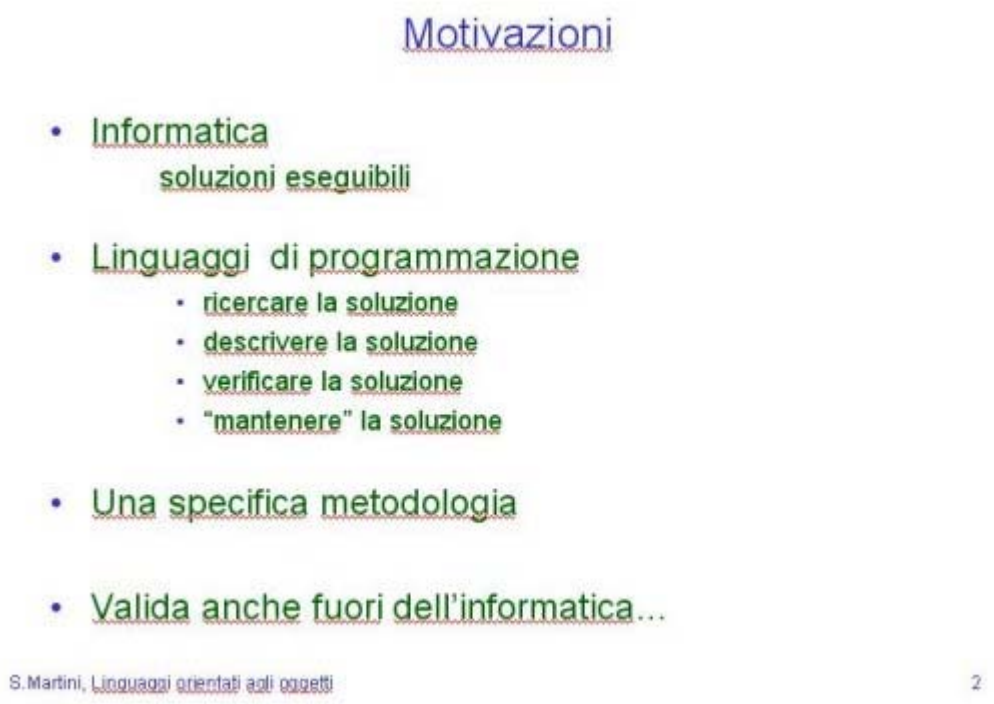
Linguaggi orientati agli oggetti
Introduzione

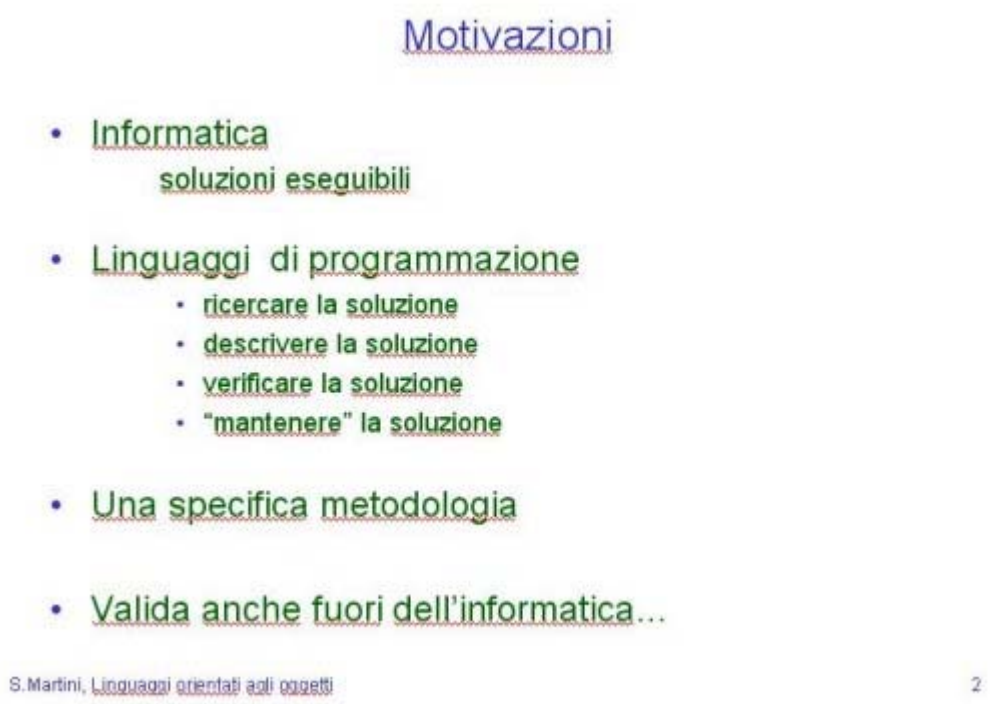
Buongiorno, sono Simone Martini, insegno linguaggi e paradigmi di programmazione all'Università di Bologna e oggi ci intratteniamo su una lezione che tratterà di linguaggi orientati agli oggetti. È chiaro che in questo modulo non possiamo entrare nei dettagli dello specifico linguaggio: non possiamo imparare a programmare in un linguaggio. Vedremo soprattutto delle tecniche metodologiche, i concetti fondamentali di questi linguaggi e perché sono utili, in generale, all'interno dell'informatica e non solo.

Motivazioni



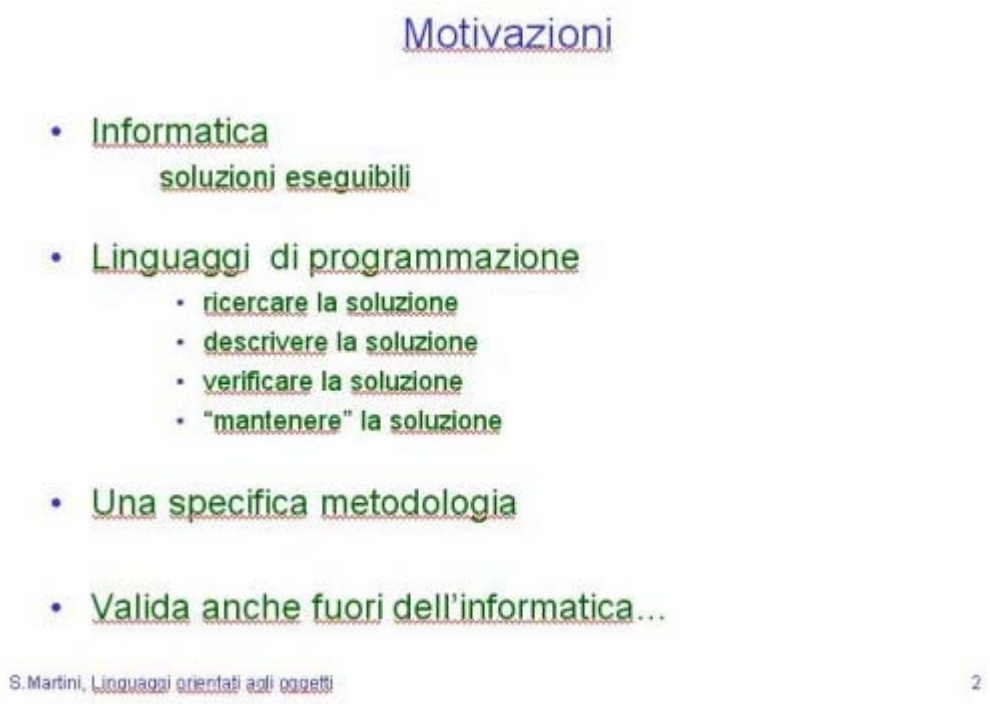
Vediamo dunque, per cominciare, alcune motivazioni. L'informatica è la scienza che costruisce delle soluzioni eseguibili. L'ingegneria costruisce manufatti, costruisce un ponte tra due argini di un fiume. La soluzione del problema è un manufatto. L'informatica costruisce delle soluzioni software, delle soluzioni virtuali. I linguaggi di programmazione sono lo strumento fondamentale con cui queste soluzioni sono prodotte. Ma sarebbe veramente riduttivo pensare che i linguaggi di programmazione servono solo per costruire la soluzione, un po' come, per riprendere la metafora di prima, il calcestruzzo serve a costruire il ponte. I linguaggi di programmazione non servono solo a costruire il programma.

Motivazioni



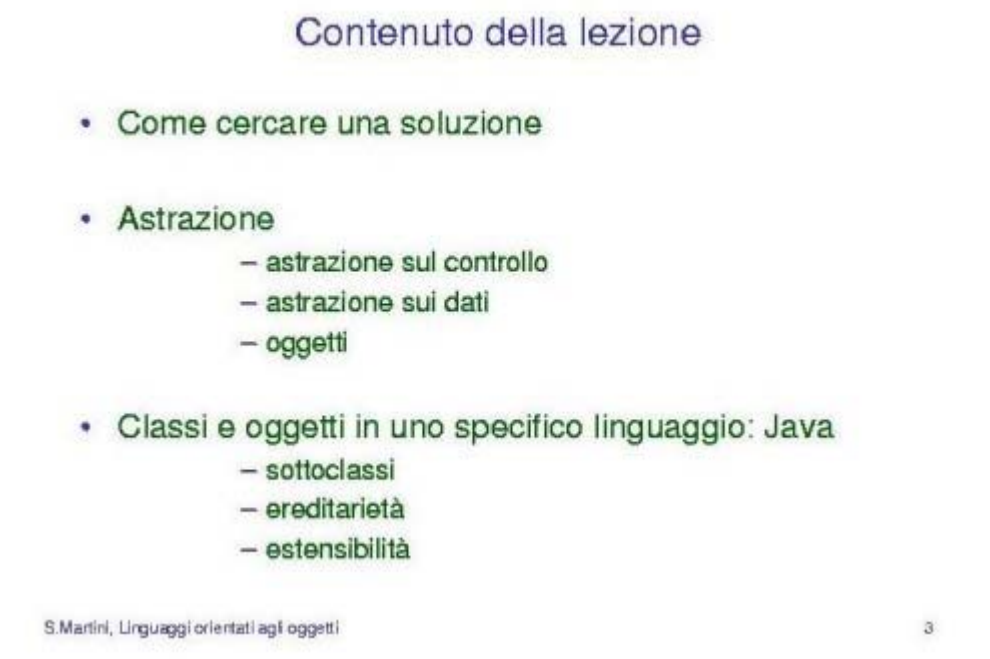
I linguaggi moderni sono uno strumento per il progettista anche per ricercare le soluzioni: devono fornire degli strumenti che facilitino la ricerca della soluzione. Ovviamente vi dovremo descrivere la soluzione, che rimane certamente la parte fondamentale del programma che viene eseguito. Ma i linguaggi delle ultime generazioni, che vanno dagli anni '60 in poi, sono pensati in modo tale che la correttezza di questa soluzione sia ragionevolmente semplice da verificare, cioè che soddisfi i requisiti per la quale è stata scritta e che non contenga errori. Infine, molto importante è il fatto che sono linguaggi progettati per far sì che il mantenimento, la manutenzione di quella soluzione non sia gravosa.

Motivazioni



Quello che vediamo in questa lezione è il modello ad oggetti che è una specifica metodologia tra quelle maggiormente in uso oggi per realizzare programmi in informatica. La vediamo sia per il suo ruolo importante all'interno dell'informatica moderna, ma anche perché è una metodologia che fornisce dei concetti che sono validi ben oltre l'informatica e utilizza strumenti per risolvere e gestire un problema complesso in modo ordinato.

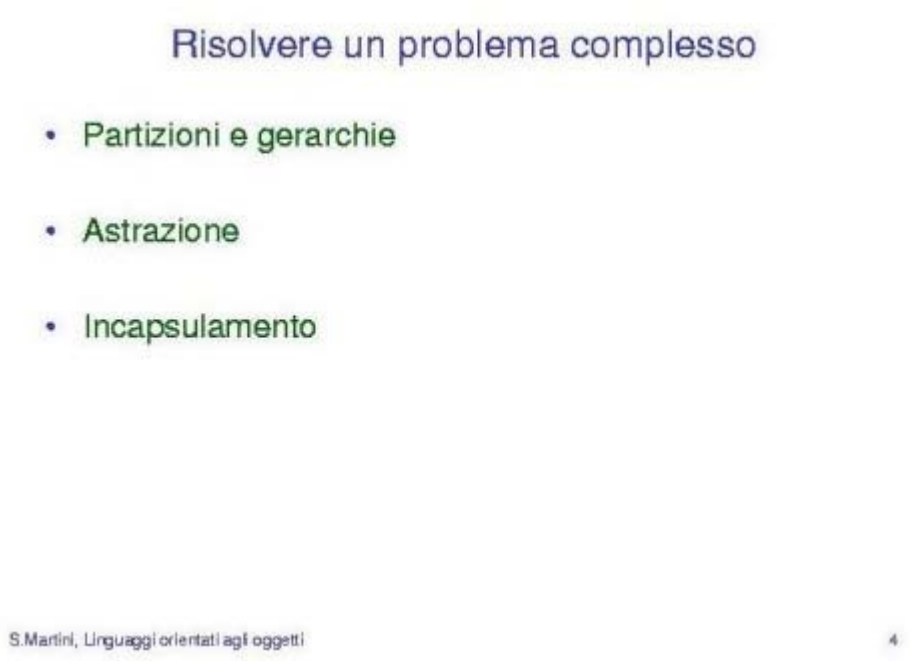
Contenuto della lezione



Ecco il contenuto della lezione. Buona parte della lezione, direi quasi la metà, sarà dedicata a quali sono le caratteristiche di una soluzione di un problema complesso e, per cercare una soluzione, dovremo parlare del concetto di astrazione. Questo è un concetto fondamentale all'interno dei

linguaggi di programmazione che l'informatica ha prodotto nel suo sviluppo teorico. Parleremo di astrazione sul controllo e di astrazione sui dati. Introdurremo poi il concetto di oggetto come un costrutto linguistico che mette insieme tutti e due i tipi di astrazione. Vedremo poi come il costrutto dell'oggetto prende la sua forma in questo tipo di linguaggio, che è il linguaggio Java, e parleremo di aspetti molto importanti tipo: le sottoclassi, l'ereditarietà, l'estensibilità e cercheremo di trarre da questo qualche conclusione.

Risolvere un problema complesso

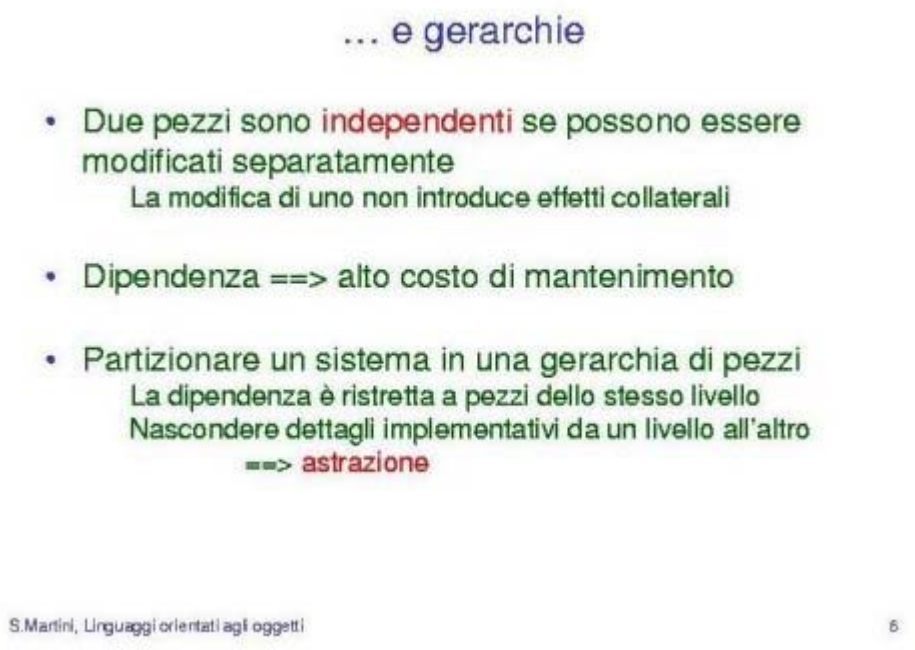


Cominciamo dal primo punto: come si cerca la soluzione ad un problema complesso. Risolvere un problema complesso richiede un notevole sforzo e capacità di invenzione. Le parole chiave di una tecnica risolutiva di un problema complesso sono: partizionare un problema, suddividerlo in livelli gerarchici e, facendo questo, astrarre aspetti importanti del problema. Deve essere un'astrazione che permetta di nascondere determinati dettagli della soluzione ad altri livelli; in termini tecnici viene chiamato "incapsulamento".

Partizioni ...

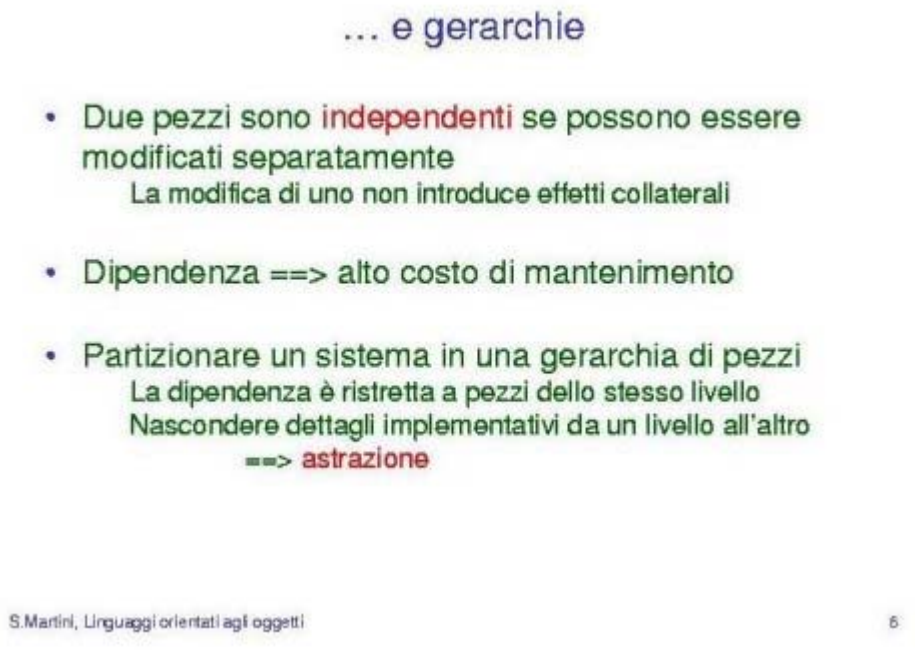
Partizioni: per risolvere un problema complesso lo si deve dividere, perché in generale, solo i problemi piccoli si risolvono bene. Si risolvono bene i problemi per i quali già si conosce la soluzione, quelli che sono varianti di un problema noto e poi ci sono quelli per i quali si vede immediatamente una soluzione. "Dividi et impera" è un vecchio motto che ogni progettista informatico deve avere sempre davanti. Un problema complesso si divide in tanti piccoli, si conquista la soluzione di ogni pezzo in modo separato e poi si rimettono insieme i pezzi. Qui, le parole separatamente e rimettere insieme sono le parole chiave di questa osservazione. Dividere è molto difficile: non c'è una tecnica che ci dice come si fa a dividere un problema in molti pezzi. Quello che sappiamo è l'obiettivo per cui dividiamo un problema: ottenere pezzi ragionevolmente indipendenti che siano ciascuno risolubile in modo indipendente dagli altri. Poi dobbiamo rimettere insieme i pezzi, ciò vuol dire farli comunicare, far scambiare loro informazioni. Ovviamente, quando i pezzi comunicano, le soluzioni che erano indipendenti ritornano in qualche modo a dipendere l'una dall'altra. La comunicazione quindi aggiunge la complessità, la quale aumenta in modo direttamente proporzionale al numero di pezzi indipendenti. In un certo senso c'è un trade off: si suddivide per avere pezzi semplici ma poi, tanto più sono i pezzi semplici, tanto più questi dovranno comunicare, quindi ci sarà un sovraccarico (un costo) per la comunicazione. Il progettista astuto è quello che trova il giusto bilanciamento tra questi due aspetti: il risparmio che si ottiene dalla divisione e il costo aggiunto dalla comunicazione.

... e gerarchie



Parlavamo di indipendenza dei pezzi, dei sottoproblemi. Due pezzi sono tra loro indipendenti se possono essere modificati separatamente. Capite che questo è un aspetto estremamente importante se vogliamo garantire che un sistema sia facilmente mantenibile. Infatti, se cambiando un pezzo, tale modifica si ripercuote su gran parte degli altri, la suddivisione in pezzi fatta risulta praticamente inutile. Ciò che vogliamo sono sottoproblemi indipendenti, cioè sottoproblemi per i quali la modifica della soluzione di uno introduca il minor numero possibile di effetti collaterali sulle soluzioni degli altri. È evidente che questo è un problema di costi: quanto maggiore è la dipendenza tanto maggiore sarà il costo della manutenzione.

... e gerarchie



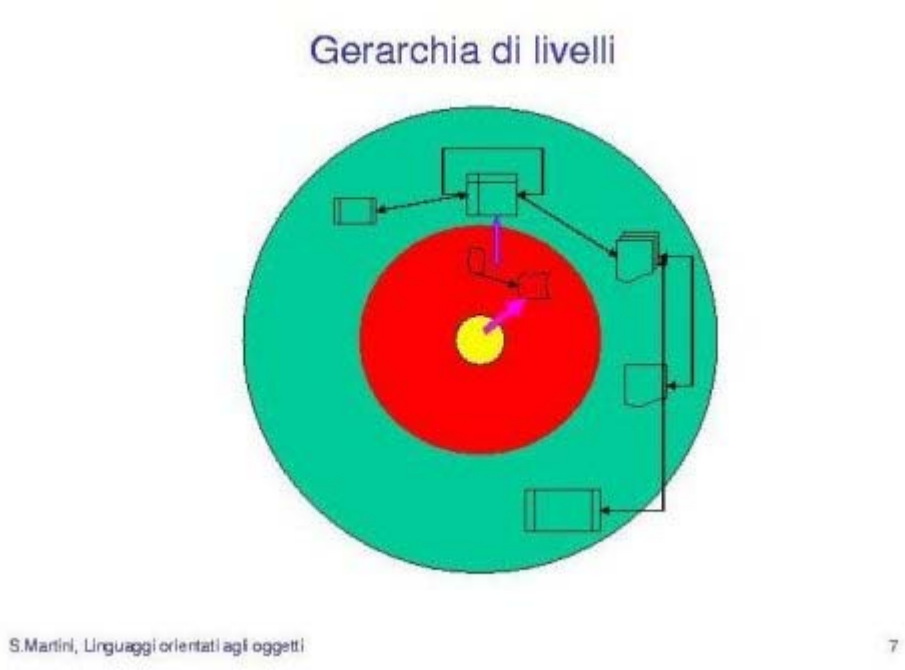
Dunque non basta partizionare: bisogna farlo in modo indipendente. Il metodo migliore, perché

spesso è il più semplice, è quello di partizionare il sistema in una gerarchia di sottoproblemi, cioè strutturare il sistema in livelli in modo tale che i dettagli della soluzione di un livello non siano visibili ad un altro livello, soprattutto a quello soprastante.

Gerarchia di livelli

Su questa trasparenza vediamo ora una sorta di esemplificazione. C'è un sistema a tre livelli rappresentati da quelle fasce colorate che si vedono. Concentriamoci sulla parte più esterna, la fascia verde. Per sommi capi un problema è suddiviso in sottoproblemi (quelle scatole che si vedono). Le soluzioni dei sottoproblemi comunicano fra di loro con delle frecce che rappresentano dei flussi informativi (esempio: variabili condivise, sincronizzazioni) tra questi sottoproblemi.

Gerarchia di livelli



Vedete che questi sottoproblemi dialogano allo stesso livello. Tutto ciò che avviene a livello inferiore è nascosto e può essere utilizzato esclusivamente attraverso delle interfacce (vedremo più avanti cosa intendiamo con questo), che sono in pratica dei modi di interazione (in figura il livello verde verso il livello rosso) che agiscono esclusivamente sulla membrana che separa questi due livelli. È evidente poi che al livello sotto utilizzeremo la stessa tecnica. Ad esempio questo pezzo, quello puntato dalla freccia di colore lilla in figura, sarà realizzato a livello inferiore come una molteplicità di pezzi e per essere risolto sarà a sua volta suddiviso in altri pezzi. Anche a livello rosso avremo una struttura di pezzi (sottoproblemi) che dialogano e interagiscono tra di loro. Qui è identificata esclusivamente quella parte di sottoproblema (i due pezzi) che corrispondono alla soluzione del problema del livello superiore, quello puntato dalla freccia lilla.

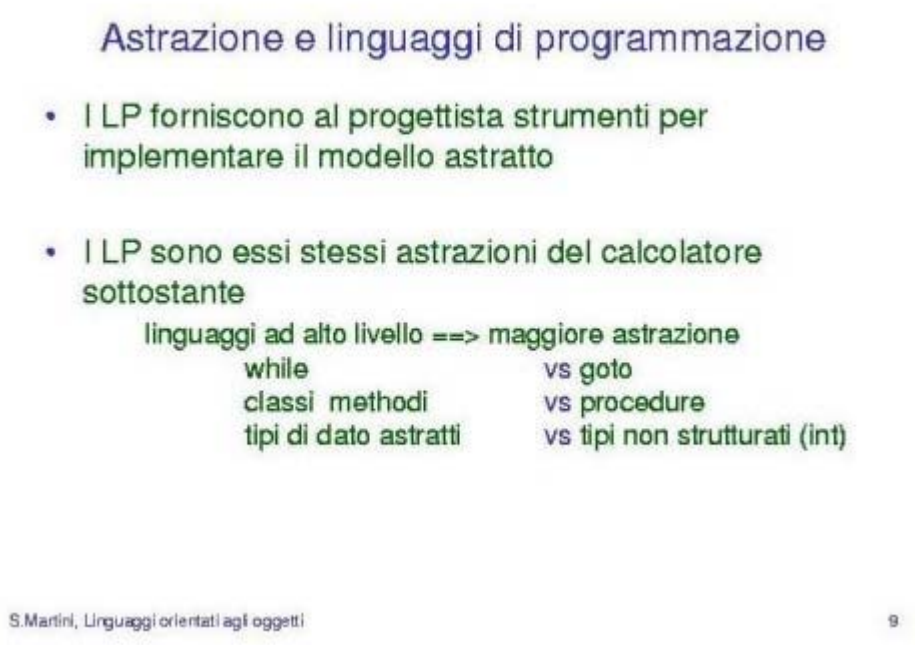
Gerarchia di livelli

La cosa essenziale è che questi livelli sono tra loro separati. Essi interagiscono esclusivamente attraverso l'interfaccia: il modulo puntato dalla freccia al livello superiore interagisce esclusivamente con i moduli che stanno sotto di lui. Non ci deve mai essere un flusso informativo diretto tra moduli del livello rosso e moduli del livello verde, perché questo introdurrebbe una complessità di gestione che renderebbe il sistema difficilmente manutenibile. Come vedete stiamo parlando ad altissimo livello, non stiamo ancora parlando di informatica, ma di risolvere problemi mediante strutturazione. Quello che l'informatica offre sono degli strumenti linguistici perché questa strutturazione, questa gerarchia di livelli, venga rappresentata all'interno di un testo che, nel caso specifico dei linguaggi di programmazione, è un testo eseguibile (un programma).

Il ruolo dell'astrazione

Ritorniamo a questo problema della gerarchia. Per garantire che i livelli siano separati, il concetto chiave è quello di astrazione. Astrarre vuol dire identificare alcune proprietà importanti di quello che si vuol descrivere, di quello che si vuole risolvere, e concentrarsi solo su queste proprietà ignorando le altre. È un procedimento ben noto della matematica e della fisica (le loro leggi sono un modello astratto della realtà) ed è una tecnica di estrema importanza all'interno della progettazione. Non ci sono tecniche standard per astrarre, ciò che è rilevante dipende certamente da ciò che si deve descrivere, ma dipende anche in modo essenziale dallo scopo del progetto. Tornando all'esempio della fisica, è evidente che il modello galileiano è utile perché fornisce una definizione a macro-livello della realtà, se si vuole andare a livello atomico quella astrazione non funziona più, ma bisogna utilizzarne altre. Questo ovviamente è vero anche all'interno dell'informatica, dove ci sono vari livelli di astrazione.

Astrazione e linguaggi di programmazione



I linguaggi di programmazione forniscono al progettista strumenti per realizzare un modello astratto e per descriverlo al livello di astrazione scelto. Un buon linguaggio di programmazione permette di descrivere un modello a vari livelli; tornando all'esempio di prima, a livello verde oppure a livello rosso. Infine un linguaggio di programmazione descrive anche l'interfaccia da eseguire. Dopo tutto, i linguaggi di programmazione sono, a loro volta, astrazioni dalla macchina su cui essi girano. Abbiamo visto nel modulo precedente, quello che descrive per sommi capi la struttura dei linguaggi di programmazione, come i linguaggi ad alto livello forniscono strumenti linguistici; ad esempio l'iterazione determinata while che è un'astrazione su un costrutto di controllo sottostante. Così, a livello di linguaggi più evoluti, come ad esempio quelli ad oggetti, ci sono dei concetti come le classi e i metodi che sono delle astrazioni su costrutti più rudimentali che sono i sottoprogrammi, o le procedure, o le funzioni.

Astrazione del controllo

Stiamo parlando di strumenti linguistici che i linguaggi di programmazione forniscono per l'astrazione. L'astrazione nei linguaggi di programmazione avviene in due sapori: il primo è quello dell'astrazione sul controllo: tipico esempio sono i sottoprogrammi (procedure o funzioni). Quello che vedete sulla trasparenza è lo schema della definizione di una procedura, di una funzione definita in Java o C. Come vedete c'è un nome P; il pezzo di programma successivo, detto corpo della funzione, serve proprio per definire tale nome P, cioè viene dato il nome P proprio a quel pezzo di programma. Il corpo interagisce con l'esterno mediante l'interfaccia, che è costituita dal parametro chiamato x, dove in questo esempio è definito come un numero intero, mentre il risultato dell'esecuzione del corpo è un numero reale, o meglio un numero razionale in doppia precisione.

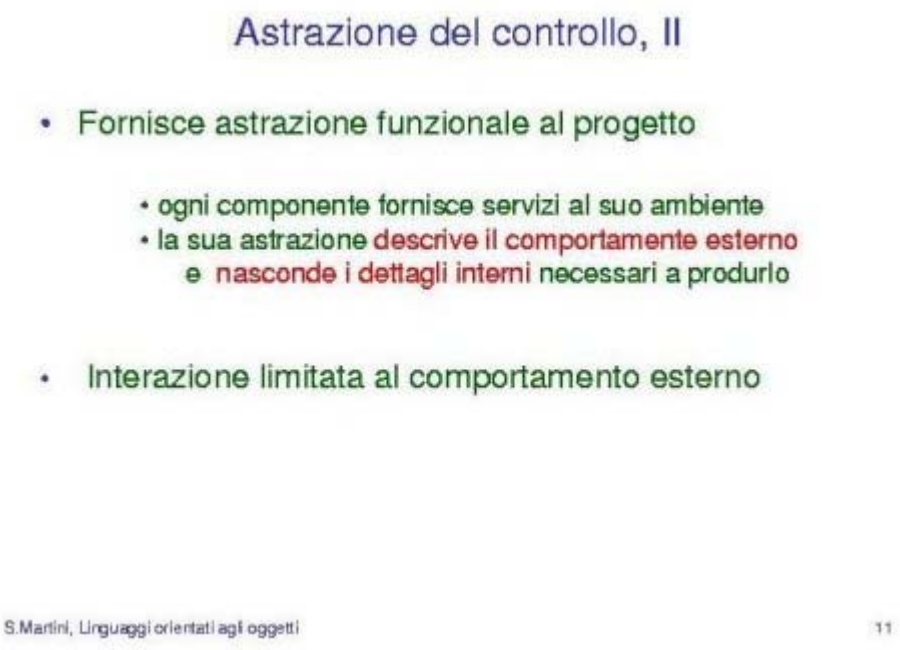
Astrazione del controllo

La cosa rilevante, che mi sta a cuore comunicarvi in questo contesto, è che qui stiamo astraendo sul corpo della funzione. Il corpo della funzione è un pezzo di programma (un pezzo di codice) che viene nascosto, dal quale si astrae mediante l'intestazione. Questa intestazione P con parametro x è tutto ciò che si vede del corpo della funzione dall'esterno di questo sottoprogramma: è un astrazione su un controllo, sul corpo della funzione. Questo vuol dire che per usare la procedura P non c'è bisogno di conoscere ciò che c'è scritto dentro il corpo della funzione, basta conoscere la sua interfaccia: il suo nome P e che parametro vuole, un numero intero.

Astrazione del controllo

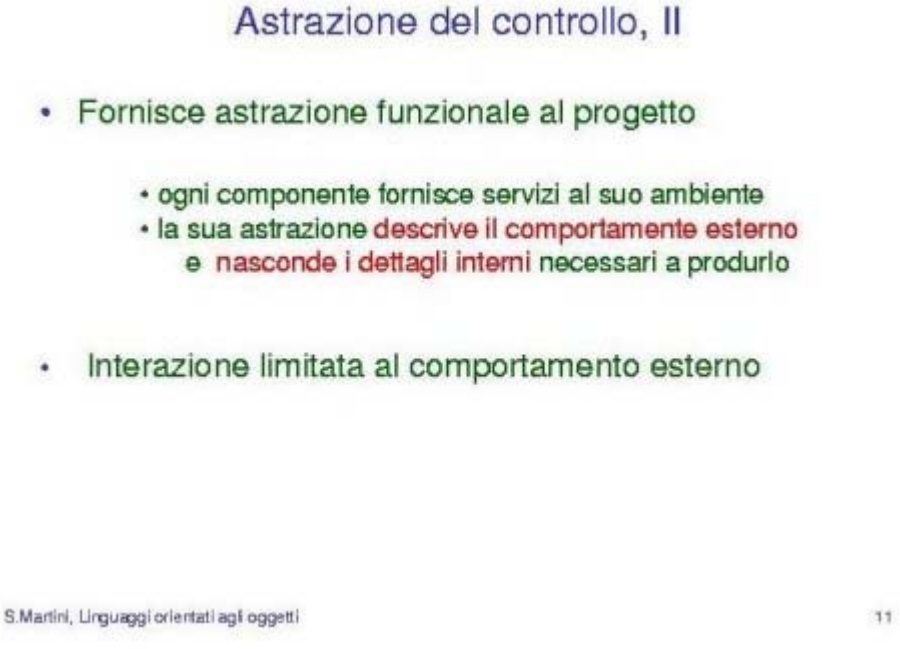
L'uso di P è indipendente dal corpo e da chi e come è stato scritto. Dualmente possiamo scrivere il corpo della funzione senza preoccuparci di dove sarà usata, né del perché sarà usata. L'unica cosa di cui dobbiamo tener è sapere bene cosa dovrà fare P, la sua specifica, e scrivere quindi il suo corpo in modo tale che P soddisfi la sua specifica. Questo è un potentissimo strumento di astrazione, che è poi assistito da altri meccanismi di questo linguaggio che sono per esempio i blocchi, costituiti dalle parentesi graffe, al cui interno è possibile definire delle variabili locali. Ad esempio z è una variabile che può essere usata nel corpo della funzione, ma che non è vista all'esterno; all'esterno di P la variabile z non esiste. Di nuovo astraiamo qualcosa dall'esterno verso l'interno.

Astrazione del controllo, II



L'astrazione sul controllo è un modo per nascondere pezzi di programma; fornisce quella che si chiama astrazione funzionale: ogni componente, in questo caso ogni procedura P, fornisce dei servizi al suo ambiente. L'astrazione descrive il comportamento esterno, il nome e i parametri che usa; nasconde dentro il blocco interno, dentro il corpo, tutti i dettagli. L'interazione tra due componenti è limitata al loro comportamento esterno: si chiamano per nome ed interagiscono tramite i parametri, senza alcun riferimento diretto tra il corpo dell'una e il corpo dell'altra. Ciò che il linguaggio di programmazione fornisce è un meccanismo per garantire che le procedure siano separate l'una dall'altra.

Astrazione del controllo, II



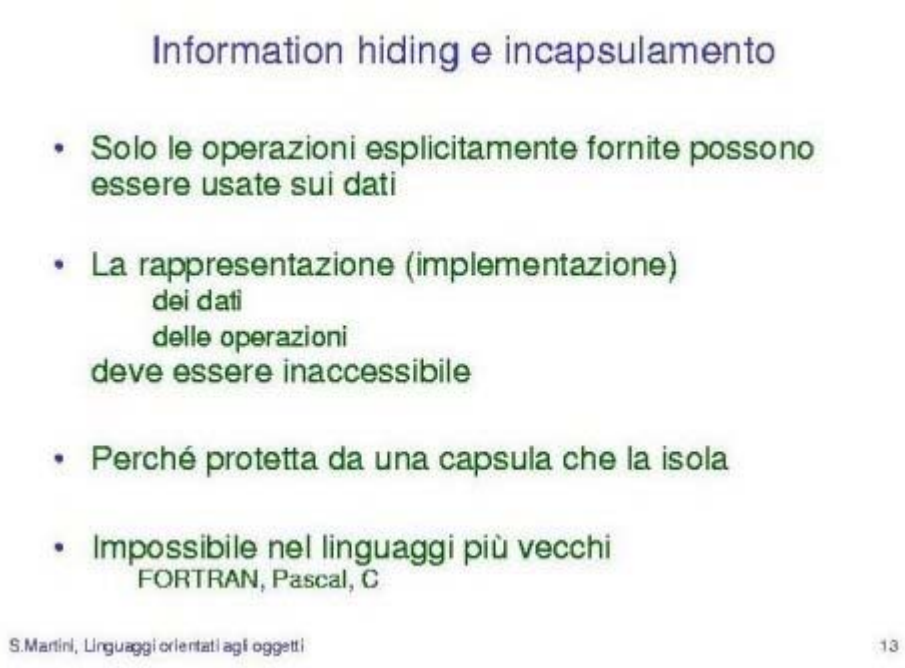
Il linguaggio di programmazione controlla che non si possa accedere all'interno di P se non

attraverso il suo nome; questi sono i meccanismi di astrazione forniti dal linguaggio. Come essi siano garantiti non possiamo affrontarlo qui (si chiamano blocchi, ambiente, ambiente statico, tipi di dato). La cosa che è importante sottolineare è che il compito fondamentale di un linguaggio di programmazione non è solo fornire le istruzioni elementari con le quali risolvere un problema, ma anche e soprattutto fornire strumenti per garantire che il programmatore utilizzi un progetto il più possibile astratto per i dati. Dicevamo che l'astrazione viene in due sapori: la prima è l'astrazione sul controllo, la seconda è l'astrazione sui dati.

Astrazione sui dati

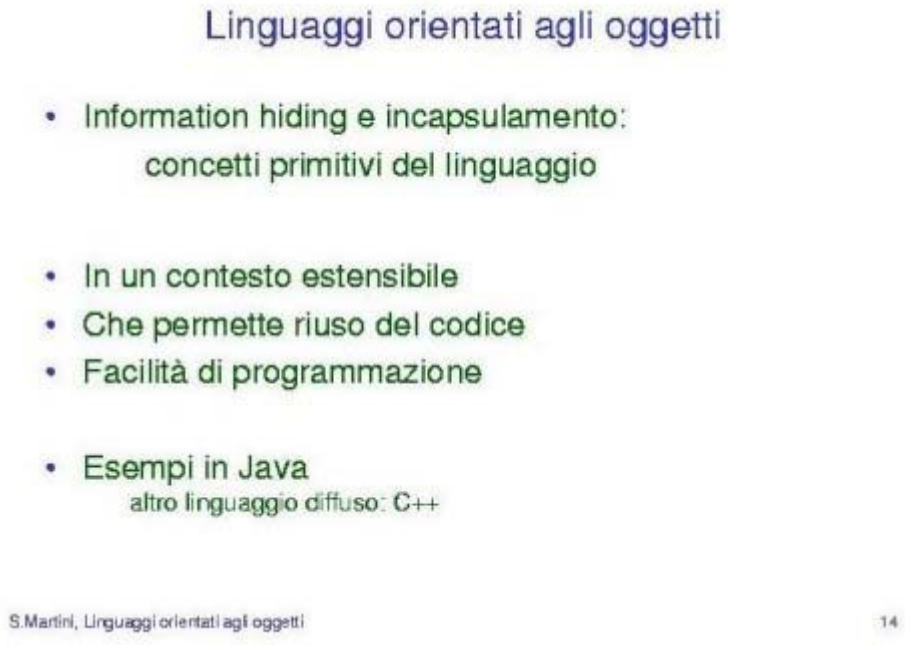
Dobbiamo introdurre brevemente il concetto di tipo di dato. Un tipo di dato è una coppia valori e operazioni. Ad esempio il tipo di dato interi (qui integer è un tipo di dato definito in Pascal) è un sottoinsieme, un intervallo finito, dei numeri interi e delle operazioni che agiscono su di essi. La cosa essenziale è che questi interi possono essere manipolati esclusivamente con quelle operazioni. Le operazioni fornite sono il solo modo per manipolare un intero, non è possibile, per esempio, utilizzare shift logici o and booleani, perché la rappresentazione degli interi viene nascosta all'utente, viene astratta dal linguaggio di programmazione. Mentre in un linguaggio di basso livello, ad esempio il linguaggio Assembler, posso accedere ad un intero come sequenza di bit, posso farci tutte le porcherie che voglio (shift logici, and logici e così via), in un linguaggio astratto, di alto livello, la rappresentazione è nascosta. Per usare qualche altra parola chiave, questa tecnica di separare l'implementazione dal suo uso si chiama nascondimento dell'informazione o information hiding.

information hiding e incapsulamento



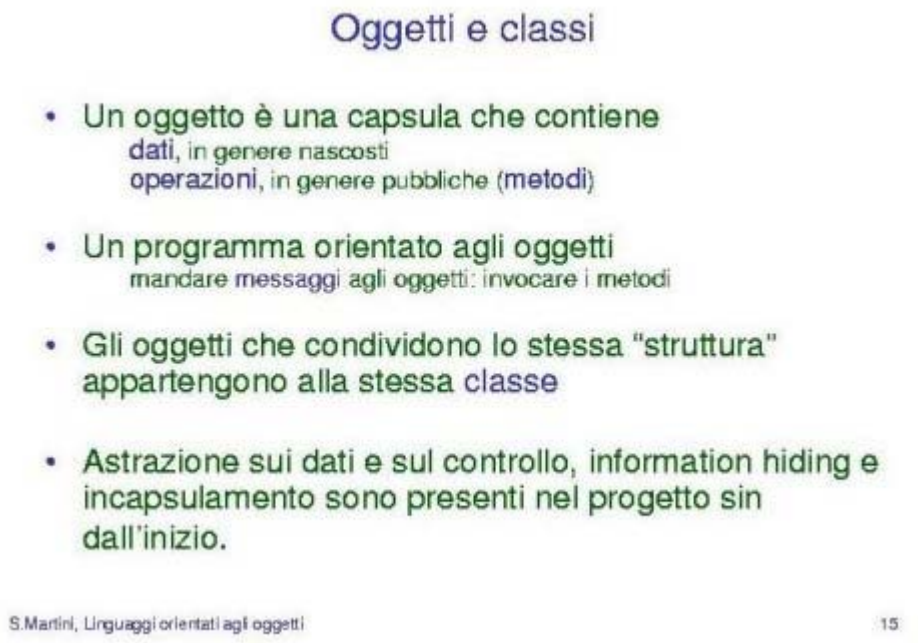
Solo le operazioni esplicitamente fornite possono essere usate sui dati; abbiamo visto l'esempio degli interi e la rappresentazione è del tutto inaccessibile all'utente. Ora, quello che noi vorremmo, è che questo non fosse possibile solo per i tipi, per le strutture predefinite del linguaggio, ma anche per quelle strutture necessarie per i sottoproblemi che l'utente definisce ed usa nel proprio programma. Vogliamo far sì che questa separazione fra uso ed implementazione sia non solo fornita per le strutture predefinite, ma l'utente possa esso stesso definire delle capsule che isolino la rappresentazione dal suo uso. Questo incapsulamento è impossibile nei linguaggi vecchi: in Fortran, in C, è largamente impossibile in Pascal, ma è possibile nei linguaggi moderni. I linguaggi orientati agli oggetti rappresentano una metodologia nella quale tale incapsulamento è realizzabile.

Linguaggi orientati agli oggetti



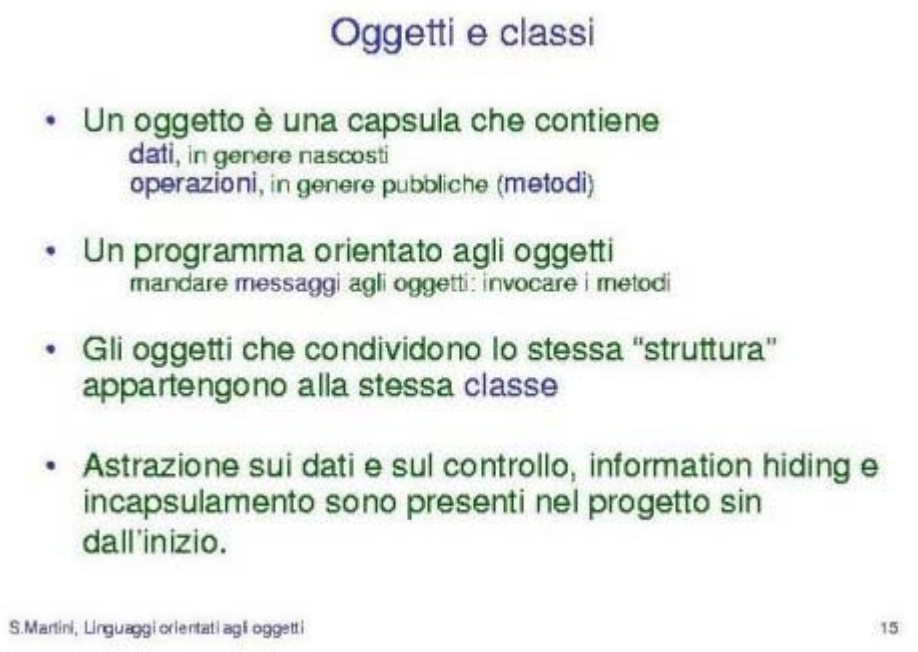
Nei linguaggi orientati agli oggetti l'information hiding e l'incapsulamento sono concetti primitivi del linguaggio stesso. Esso è costruito intorno a costrutti linguistici per garantire il nascondimento dell'informazione. Tutto questo è possibile anche in altri linguaggi non orientati agli oggetti. La cosa fondamentale che contraddistingue i linguaggi orientati agli oggetti è che questo avviene in un contesto estensibile, cioè una volta incapsulato un oggetto, questo non è una capsula scolpita nella pietra che non può essere più modificata, ma una capsula che può essere estesa, permettendo il riuso del codice e la facilità di programmazione. Questa è la chiave di volta della tecnologia orientata agli oggetti che la differenzia dalle altre tecniche di programmazione che riescono a garantire l'incapsulamento. Noi vedremo qualche esempio nel linguaggio Java, che è uno dei più diffusi; l'altro linguaggio ampiamente usato orientato anch'esso agli oggetti è il C++, ma per eleganza di progetto, per semplicità d'uso, per tutta una serie di motivi che forse non è il caso di elencare, Java è il linguaggio più indicato per introdurre la programmazione orientata agli oggetti.

Oggetti e classi



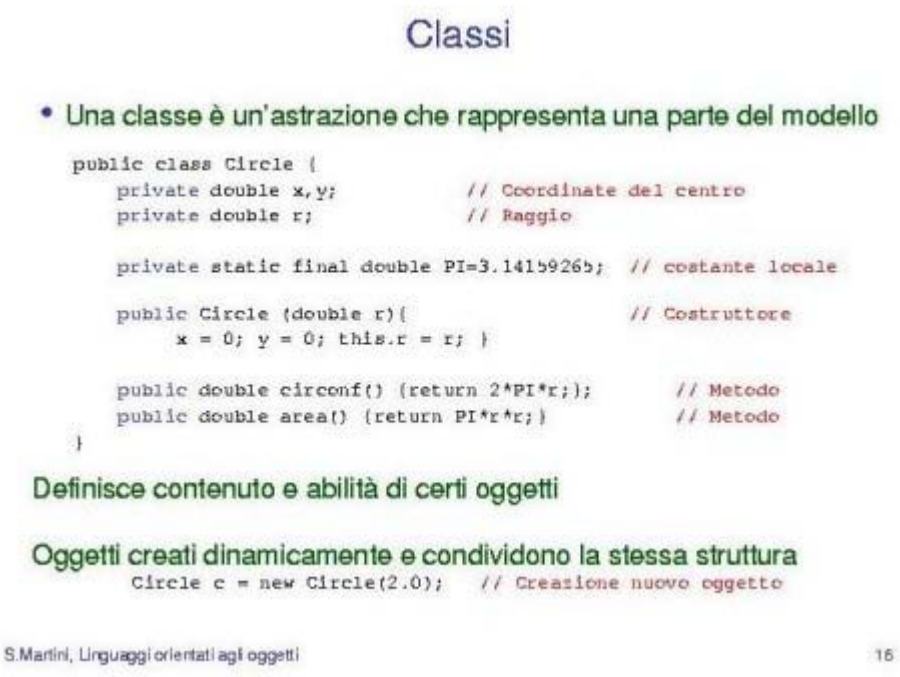
Scendiamo un po' nel dettaglio di cosa sono gli oggetti e introduciamo un nuovo concetto che è quello di classe. Abbiamo già detto che la cosa fondamentale è l'incapsulamento e un oggetto è infatti una capsula che contiene due cose: i dati e le operazioni su quei dati. Un oggetto è quindi qualcosa di unitario che garantisce nello stesso momento sia l'astrazione sul controllo, sia l'astrazione sui dati. Questi due tipi di astrazione vengono quindi fusi in un oggetto, che astrae sui dati che in genere sono nascosti e sulle operazioni che possono agire su quei dati. In generale sono pubblici i nomi e l'uso delle operazioni, mentre invece i dati sono spesso totalmente nascosti. Nel gergo object-oriented le operazioni si chiamano metodi. In sostanza un oggetto incapsula dei dati, sui quali si può agire attraverso dei metodi.

Oggetti e classi



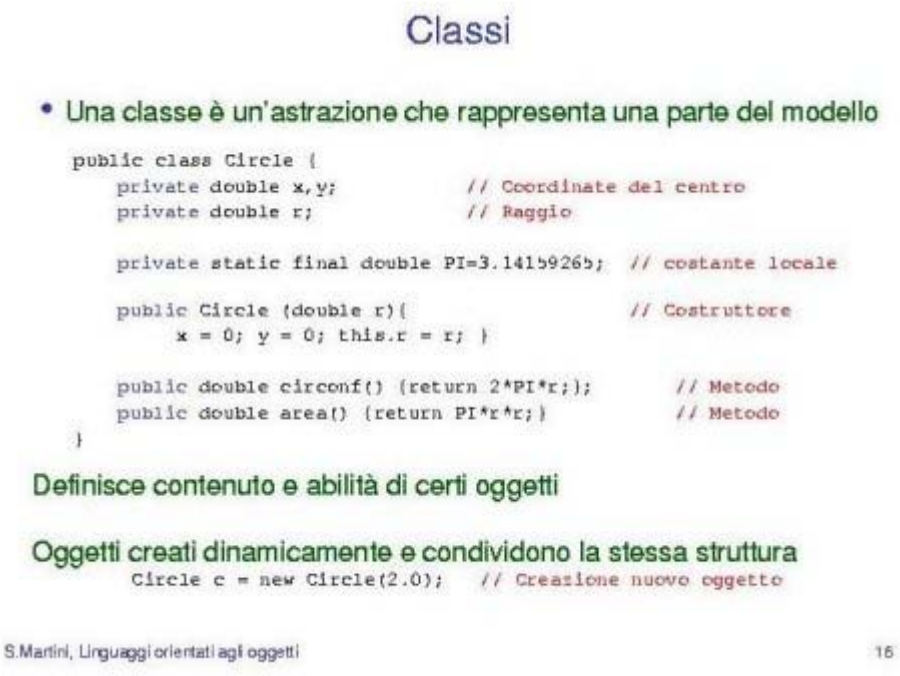
Come vengono invocati questi metodi? Come possiamo far sì che un metodo agisca sui dati? Ciò viene fatto inviando un messaggio all'oggetto. Un programma orientato agli oggetti è una collezione di oggetti che comunicano mediante messaggi. Un messaggio è la richiesta di invocazione di un metodo: un oggetto A chiede ad un oggetto B, mediante un messaggio, di eseguire un metodo di B sui dati di B e di restituire il risultato ad A. Ora, questi oggetti non costituiscono una collezione caotica, molti di essi sono simili tra loro, in particolare, se condividono una stessa struttura, si dice che appartengono alla stessa classe. In realtà la progettazione avviene in modo inverso, cioè vengono prima definite alcune strutture comuni a più oggetti, le classi appunto, e poi vengono creati gli oggetti a partire dalla classe. Prima definiamo la struttura del nostro mondo, definendo varie classi e poi si creano gli oggetti che popolano questo mondo, partendo dalle classi, dalla struttura che questi oggetti devono avere.

Classi



La classe, dal punto di vista progettuale è un'astrazione che rappresenta una parte del modello. Qui abbiamo, nella trasparenza, un esempio di una classe definita nel linguaggio Java. Una classe che abbiamo chiamiamo circle e che rappresenta un cerchio. Cominciamo col vedere come è strutturata questa classe. Abbiamo già detto che un oggetto è un'astrazione sia sui dati che sul controllo. Una classe è la definizione della struttura di molti oggetti. Innanzitutto vediamo che la classe considerata è composta di due parti, la parte superiore (le prime tre righe della trasparenza, quelle contraddistinte dalla parola private) è la parte che specifica i dati, l'astrazione sui dati.

Classi



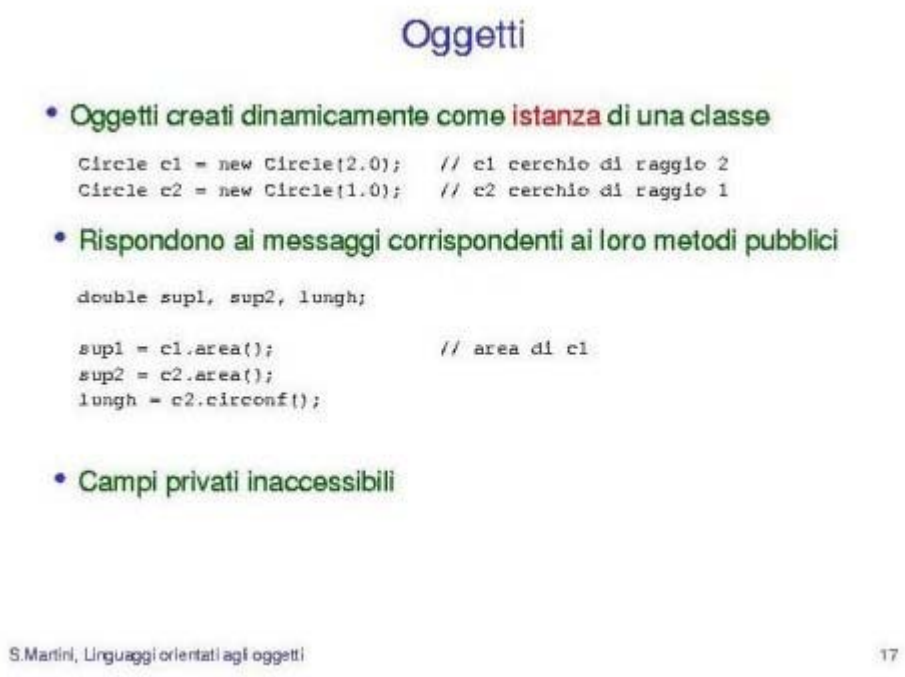
La seconda parte (le seconde tre linee della classe, quelle contraddistinte dalla parola public) sono le operazioni, i metodi, cioè l'astrazione sul controllo. Vediamo ora i dati: sono due variabili di tipo

reale x, y, un'altra variabile r e un altro nome p che viene fissato al valore 3.1415. Un cerchio in questa astrazione viene quindi individuato da una coppia di reali x e y, che interpretiamo come le coordinate del centro del cerchio, e un reale che è il raggio del cerchio. Fa parte dei dati di questo cerchio anche una costante p (= p-greco). Quali operazioni manipolano questi dati? Sono tre: l'operazione circle che costruisce un cerchio. Vuole come parametro un numero reale r e costruisce un cerchio di centro l'origine degli assi cartesiani e di raggio r. Tale metodo, in gergo object-oriented, viene detto costruttore, perché serve a inizializzare (a costruire) un oggetto, che è appunto un cerchio che ha centro nell'origine degli assi e raggio r, quest'ultimo fornito al momento della creazione.

Classi

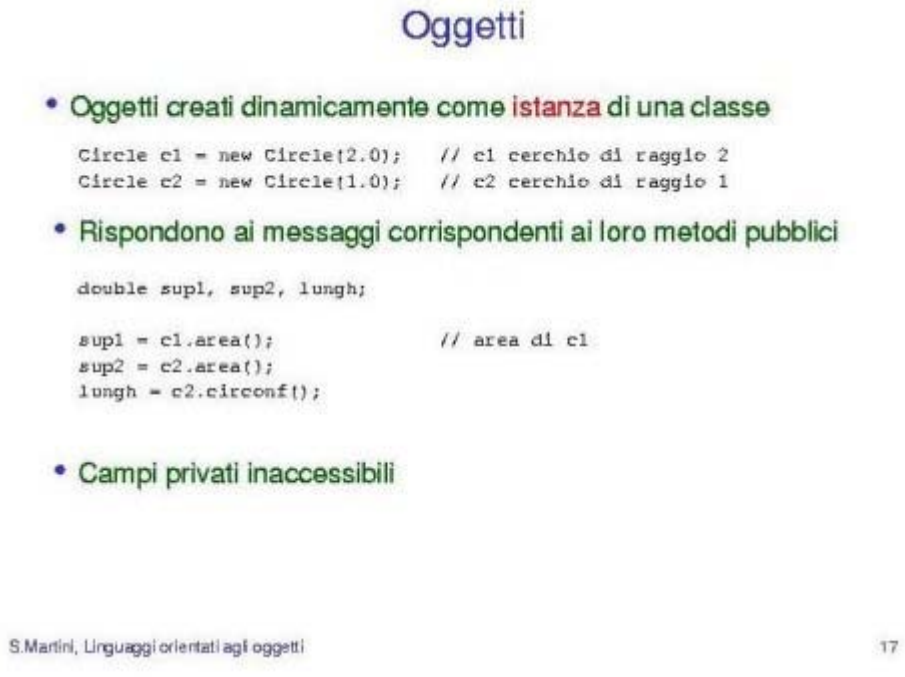
Ci sono poi altri due metodi: circonf, che restituisce 2*pi-greco*r (la circonferenza di quel cerchio su cui agirà questo metodo) e area, che ritorna pi-greco*r^2 (restituisce l'area di quel cerchio su cui questo metodo agirà). Senza entrare in troppi dettagli, osservate come la parte dei dati è stata contraddistinta dalle parole private, che non sono indispensabili ma in questo caso stabiliscono che quei dati sono inaccessibili dall'esterno, sono effettivamente astratti, sono nascosti dall'esterno della classe. Mentre invece i metodi (il costruttore circle, Circonf, Area) sono pubblici, cioè accessibili dall'esterno. C'è quindi un nascondimento totale dei dati ed una pubblicità dei metodi, che ovviamente si limita solo al loro nome e parametri; non si potrà mai vedere all'esterno cosa si trova dentro il corpo di questi metodi. Una classe definisce: la struttura, il contenuto, le capacità e le abilità di determinati oggetti.

Oggetti



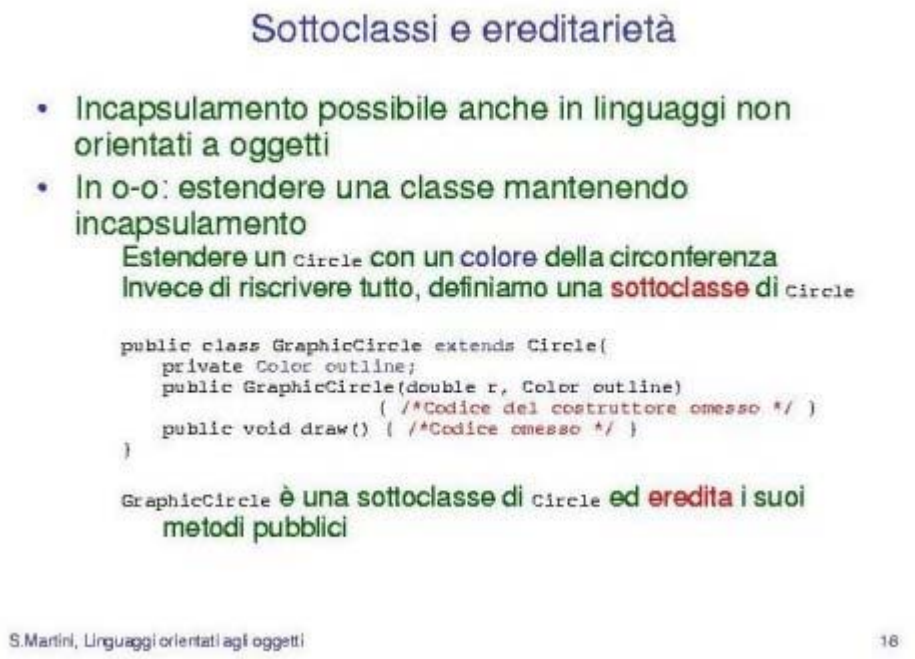
Gli oggetti vengono creati dinamicamente, condividono tutti la stessa struttura, quella fissata dalla classe. Vediamo in questo esempio (nell'ultima riga della trasparenza) che viene costruito un nuovo cerchio di nome c e ciò viene fatto a partire dal costruttore circle con parametro 2. c sarà quindi un oggetto che ha questa struttura: avrà al suo interno quattro numeri reali (compreso pi-greco) e tre metodi per agire su di esso. In particolare c è stato creato col parametro 2, quindi è un cerchio con centro nell'origine degli assi e raggio 2. Questi dati non sono accessibili dall'esterno, l'unica cosa che c ci forniscono è la capacità di rispondere a questi metodi: il metodo circle che lo ha costruito, il metodo circonf che ci restituisce la circonferenza di questo cerchio di raggio 2 e il metodo area che ci restituisce l'area di questo cerchio di raggio 2.

Oggetti



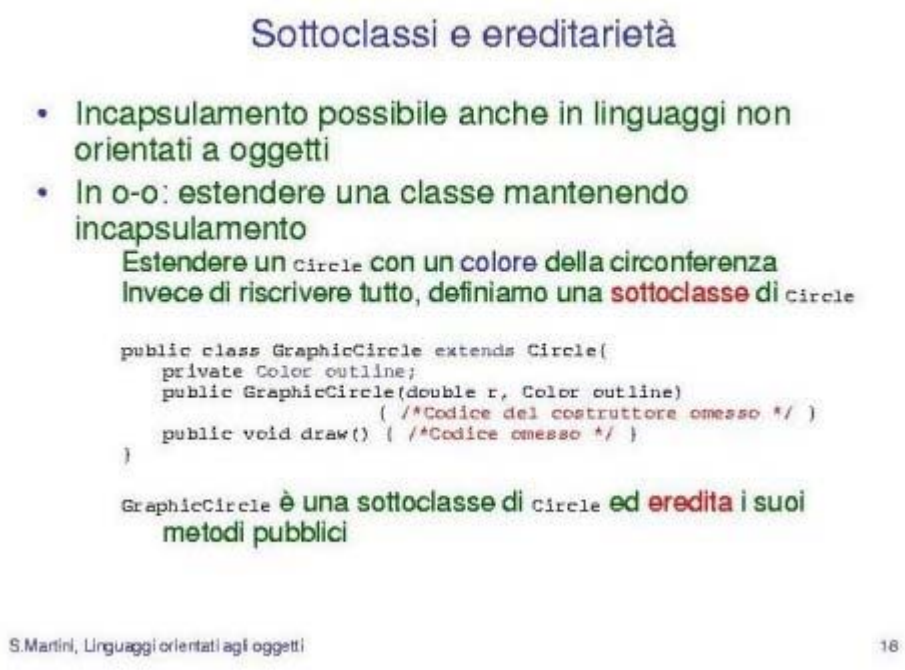
Gli oggetti sono creati dinamicamente come istanza di una classe. Abbiamo creato prima l'oggetto c, ora creiamo c1 e c2. c1 è un altro cerchio di raggio 2, ha le stesse caratteristiche di c, è diverso da c perché ha un nome distinto ed è stato creato come nuova istanza. c2 è un altro cerchio di raggio 1. Quindi, in questo momento, il nostro mondo è popolato da tre oggetti, tutti con la stessa struttura, ma diversi come valori: c e c1 sono cerchi di raggio 2, c2 è un cerchio di raggio 1. Questi cerchi, o oggetti, rispondono ai messaggi che corrispondono ai loro eventi pubblici; qui abbiamo due esempi, c1 risponde al metodo area. Questa sintassi, che vedete nella trasparenza ,c1.area sta ad indicare che posso mandare il messaggio corrispondente al metodo area a c1. Significa che l'oggetto c1 risponde al metodo area restituendo la propria area. Lo stesso possiamo fare con c e c2 inviando lo stesso messaggio a questi altri due oggetti. c, c1 e c2 rispondono quindi agli stessi metodi, ovviamente con risultati diversi: c1.area ci da l'area di un cerchio di raggio 2, c2.area ci da l'area del cerchio c2 che ha raggio 1. I dati sono del tutto inaccessibili all'utente, il nascondimento dell'informazione è in questo caso perfetto.

Sottoclassi e ereditarietà



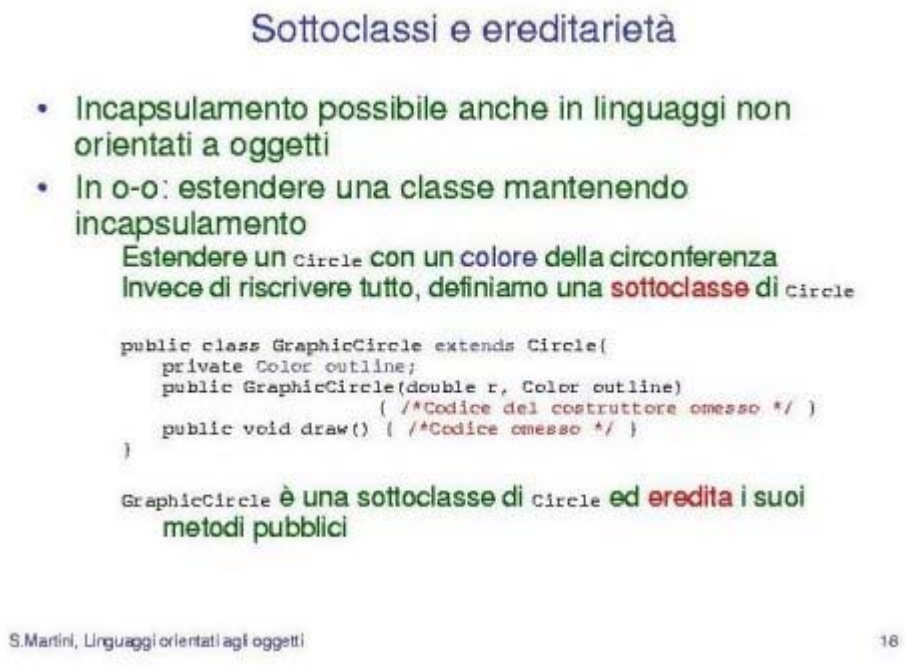
L'incapsulamento, che è uno dei concetti sui quali abbiamo insistito maggiormente fino adesso, è possibile in linguaggi che non sono orientati agli oggetti. Ciò che invece caratterizza la metodologia orientata agli oggetti è la possibilità di estendere una classe mantenendo l'incapsulamento. Se si ha una classe alla quale voglio aggiungere delle funzionalità non ho bisogno di rompere la capsula, aggiungere le funzionalità e ricostruire una nuova capsula. Ciò risulterebbe una cosa terribile: nel momento in cui apro una capsula per ricostruircene un'altra intorno, do la possibilità di accedere al suo contenuto. Quello che è possibile fare con i linguaggi object-oriented è di prendere una classe ed estenderla, cioè aggiungerci delle funzionalità in una nuova capsula, senza aprirla. Questo viene fatto attraverso il concetto di sottoclasse, o estensione di una classe.

Sottoclassi e ereditarietà



Vediamo un esempio. Vogliamo estendere un cerchio aggiungendo un colore alla sua circonferenza. Immaginando che i cerchi possano essere disegnati sullo schermo; non hanno solo un centro e un raggio ma hanno anche un colore della circonferenza. Potremmo definire una nuova classe che oltre ad x, y e r abbia anche un colore. Ciò è ovviamente possibile, ma non è un buon modo di procedere. Un modo corretto per risolvere questo problema è di estendere il cerchio. Creiamo una nuova classe, chiamata GraphicCircle come una classe che estende circle.

Sottoclassi e ereditarietà



Estendere vuol dire che mantiene tutto ciò che c'è in circle, aggiungendo alcune nuove proprietà. Un GraphicCircle è tutto quello che è un circle, sia le variabili (x,y,r) sia i metodi (circle, Area, Circonf), più un nuovo dato che io ho chiamato outline, che è di tipo Colore, più un metodo GraphicCircle che

è di tipo costruttore. Costruirà un oggetto della classe GraphicCircle, che avrà come parametri un raggio e un colore, che saranno appunto il raggio e il colore dell'oggetto creato. Abbiamo infine un altro nuovo metodo chiamato draw che serve per disegnare questo cerchio su un piano cartesiano.

Sottoclassi e sottotipi

Osservate come GraphicCircle è definita come una estensione di circle: GraphicCircle è una sottoclasse di circle. In gergo si dice che eredita tutti i metodi pubblici di circle; ciò vuol dire che un oggetto GraphicCircle sa rispondere non solo al metodo draw, ma sa rispondere anche al metodo area e al metodo circonf. Questo non è esplicitato, ma discende dal fatto che GraphicCircle è un'estensione di circle. Nella prossima trasparenza c'è il riassunto di quanto abbiamo appena detto: una sottoclasse è definita come una estensione, ogni oggetto della classe GraphicCircle è anche un oggetto della classe circle. Potremmo quindi usare un cerchio grafico ogni volta che viene richiesto un semplice cerchio, semplicemente ci dimentichiamo del fatto che ha anche caratteristiche in più. Questa è una dote di grandissima flessibilità.

Sottoclassi e ereditarietà: esempio

Sottoclassi e ereditarietà: esempio

```
public class GraphicCircle extends Circle{
    private Color outline;
    public GraphicCircle(double r, Color outline)
                    {/*Constructor code omitted */}
    public void draw() {/*Code omitted */}
}

GraphicCircle gc = new GraphicCircle(3,"blue");

double a = gc.area();       // gc è un Circle !

gc.draw();                  // gc è un GraphicCircle

Circle c = gc;      // c è gc "visto come" un Circle
a = c.area();

c.draw();           // Errore! Non c'è un metodo draw in Circle
```

S.Martini, Linguaggi orientati agli oggetti 20

Vediamo un esempio di sottoclassi un po' più esteso. Viene mantenuta la stessa struttura, tra parentesi ho lasciato la definizione della classe GraphicCircle, che è la stessa che abbiamo discusso precedentemente. Innanzitutto possiamo creare un cerchio grafico chiamato gc. Per far questo ci serviamo del suo costruttore GraphicCircle e lo definiamo di raggio 3, di colore blu e centro nell'origine degli assi. Possiamo ora mandare dei messaggi a gc. Ad esempio, inviando un messaggio del tipo gc.area, gc riesce a rispondere in quanto appartiene anche alla classe circle, essendo un oggetto di una sottoclasse di circle. Naturalmente, essendo un GraphicCircle, saprà rispondere anche al metodo draw che lo disegnerà nello schermo.

Sottoclassi e ereditarietà: esempio

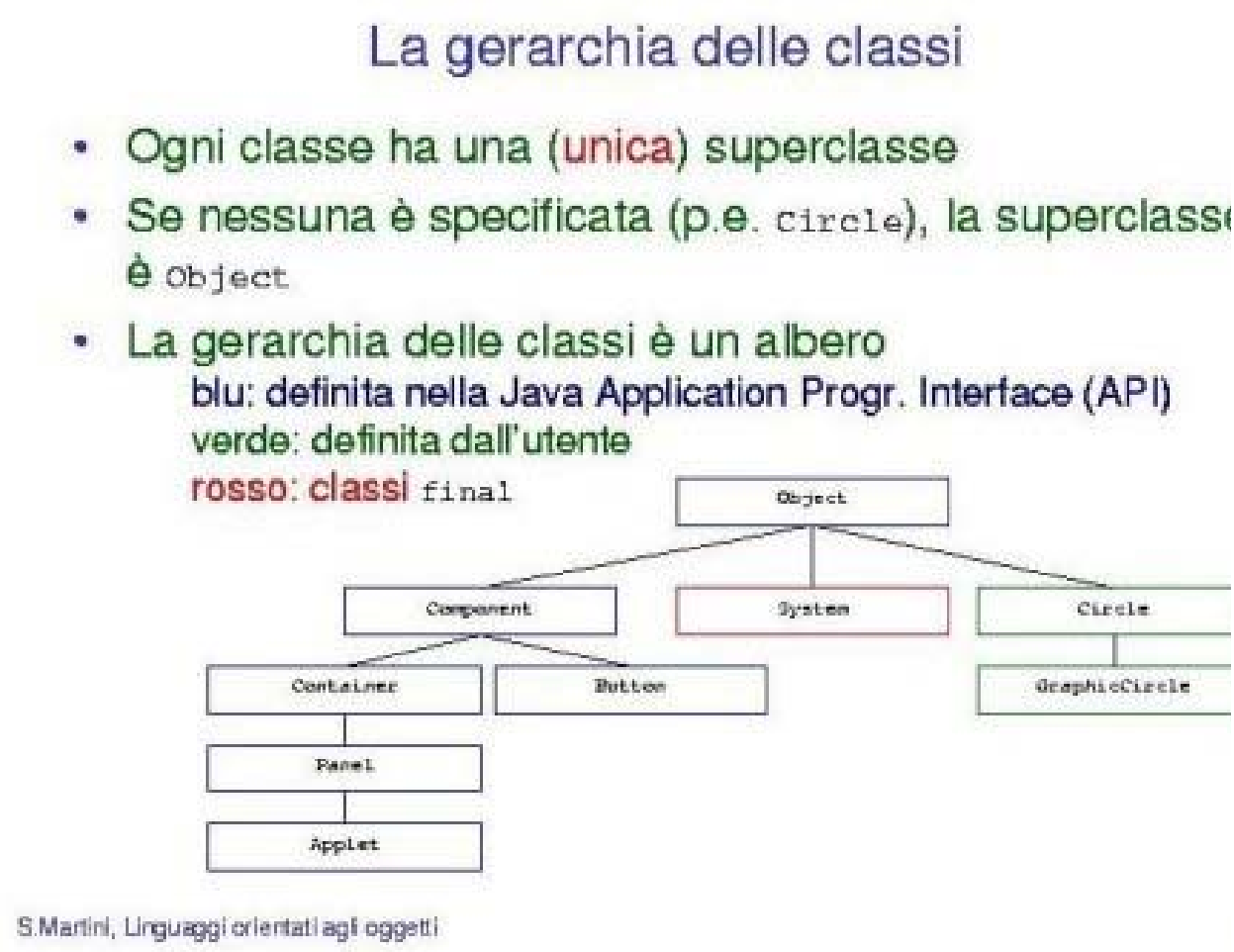
Sottoclassi e ereditarietà: esempio

```
public class GraphicCircle extends Circle{
    private Color outline;
    public GraphicCircle(double r, Color outline)
                    {/*Constructor code omitted */}
    public void draw() {/*Code omitted */}
}

GraphicCircle gc = new GraphicCircle(3,"blue");

double a = gc.area();       // gc è un Circle !

gc.draw();                  // gc è un GraphicCircle

Circle c = gc;      // c è gc "visto come" un Circle
a = c.area();

c.draw();           // Errore! Non c'è un metodo draw in Circle
```

S.Martini, Linguaggi orientati agli oggetti

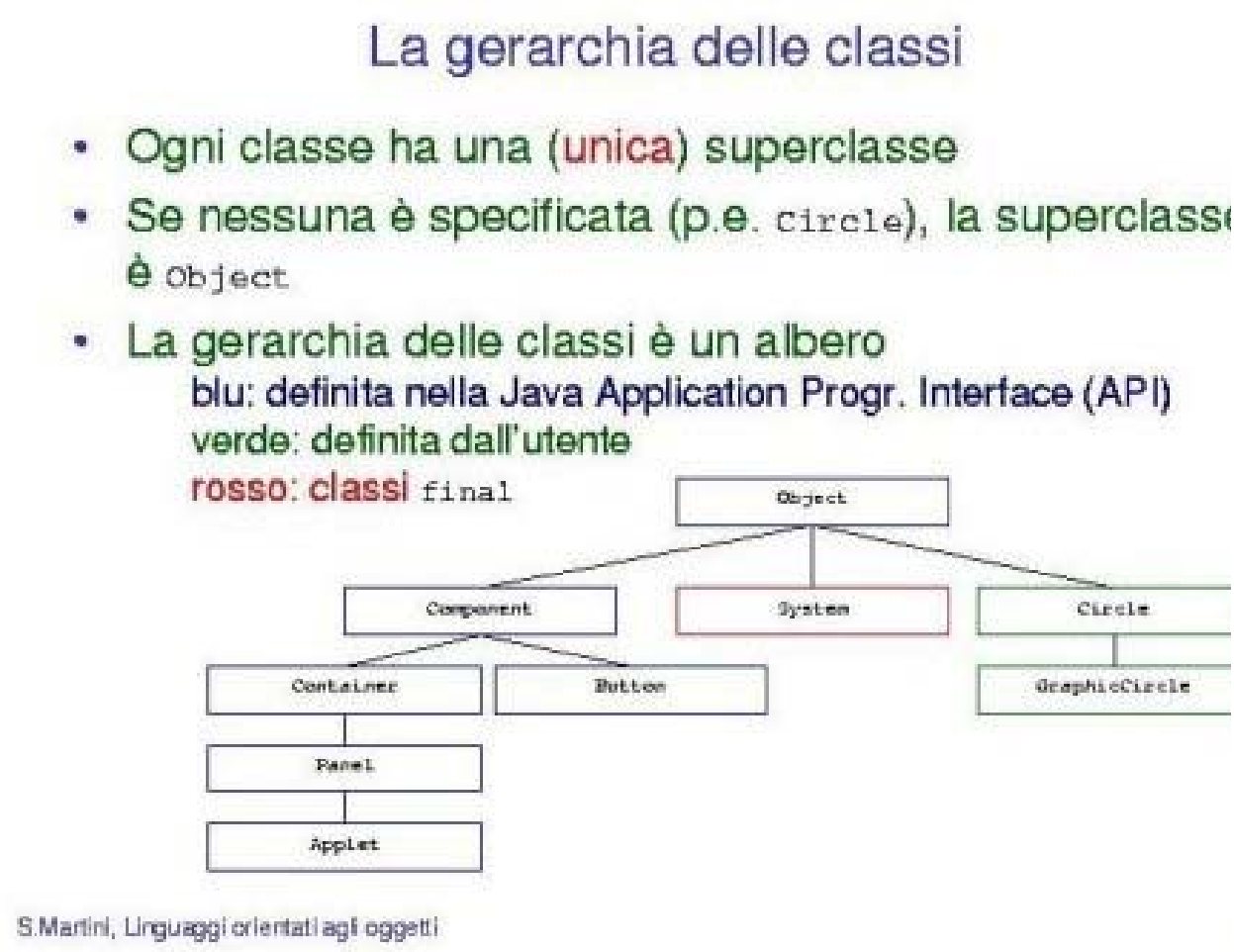
Possiamo definire c come un altro nome per gc, ma portando gc ad essere soltanto un cerchio e non un cerchio grafico. Pertanto c potrà ricevere dei messaggi soltanto come cerchio, risponderà per esempio al metodo area, ma non al metodo draw. Se viene chiesto a c di eseguire il metodo draw viene generato un errore, che sarà segnalato dal traduttore, dal controllore sintattico del linguaggio, prima che il programma possa venire eseguito e segnalerà appunto che draw non è un metodo che c può eseguire. Quindi c, sebbene sia un altro nome di gc, viene visto soltanto come cerchio, perché dichiarato nella classe circle e quindi non riesce a vedere alcuna proprietà o metodo di GraphicCircle.

La gerarchia delle classi



Abbiamo visto questo semplice esempio di sottoclasse (la classe GraphicCircle sottoclasse di circle), spostiamoci ora ad un contesto più ampio. Le classi in Java hanno una struttura molto elegante perché ogni classe può avere un'unica superclasse. Questo fa sì che la gerarchia delle classi in Java sia un albero, dove ogni classe che non ha alcuna sopraclasse (nel nostro esempio era la classe circle) viene considerata dal linguaggio come una sottoclasse della classe object, che è una generica classe definita dal linguaggio e che costituisce quindi la radice dell'albero della gerarchia tra classi.

La gerarchia delle classi



La gerarchia delle classi è un albero; abbiamo delle gerarchie che sono quelle definite dall'utente ad esempio circle, che è una sottoclasse di object per definizione, poi GraphicCircle che è una sottoclasse di circle perché così l'abbiamo definita. Ci sono poi delle gerarchie che fanno parte della definizione del linguaggio, ad esempio Component che ha come sottoclassi Button (bottone che serve per realizzare delle interfacce utente) oppure Container e così via. Ci sono poi delle classi che sono definite finali e sono le classi che non possono essere estese, cioè si fissa al momento della loro definizione che non possono avere sottoclassi. La struttura della gerarchia è abbastanza complicata, questa che vedete è una porzione dell'albero estremamente ridotta, ma quest'albero è estensibile dall'utente.

Sottoclassi e ereditarietà: altro esempio (I)

Sottoclassi e ereditarietà: altro esempio (I)

```
class Time {
  private int hour,min;

  public Time(int h,int m){                    //Costruttore
       hour = h; min = m; }

  public int getHour(){return hour;}          // Metodo
  public int getMin(){return min;}            // Metodo

  public void addMin (int numMin){
       min = min + numMin;
       while (min>59) {min = min - 60;
                       hour = hour + 1; } ;
       while (hour > 23) hour = hour - 24;
  }
}
```

S.Martini, Linguaggi orientati agli oggetti

Sulle trasparenze che trovate nel materiale, è descritto un altro esempio che io non sto a discutere in dettaglio. Abbiamo una classe Time che introduce dei dati privati, un tempo in ore e minuti. Avremo quindi un metodo costruttore che assegnerà per ogni nuovo oggetto una certa ora e minuti e dei metodi che permettono di leggere il campo ora o il campo minuti e che consentono di aggiungere un certo numero di minuti all'oggetto. Capite che per aggiungere i minuti non basta sommare i numeri che si vogliono aggiungere ai minuti dell'oggetto, bisogna naturalmente fare tale somma modulo 60, cioè occorrerà aumentare l'ora nel caso in cui il valore dei minuti sia diventato maggiore di 60.

Sottoclassi e ereditarietà: altro esempio (II)

Sottoclassi e ereditarietà: altro esempio (II)

```
public class TimeSec extends Time{
        private int sec;

        public TimeSec(int h,int m,int s){          //Costruttore
                super(h,m); sec =s; }

        public int getSec(){return sec;}            //Metodo

        public void addSec (int numSec){
                sec = sec + numSec;
                while (sec>59) {sec = sec - 60;
                                this.addMin(1);}
                }
        }
```

S.Martini, Linguaggi orientati agli oggetti

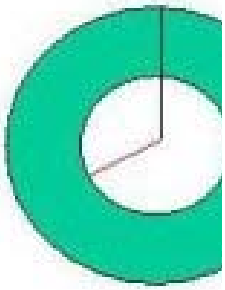
Questa classe può essere estesa, in questo caso estendiamo Time con una nuova classe: TimeSec. Le istanze sono ancora dei tempi, dove abbiamo indicati, oltre ai minuti e alle ore, anche i secondi. In tal caso avremo un nuovo costruttore che avrà in ingresso un certo valore per le ore, un certo valore per i minuti e uno per i secondi ed inizializza un nuovo oggetto con quei valori dati in input. Abbiamo usato una parola importante in Java, che è super: permette di invocare il costruttore di quella classe. Abbiamo anche in questo caso, come per Time, un metodo per leggere il tempo, ed un nuovo metodo che aggiunge i secondi.

Estensibilità: ridefinire un metodo

Estensibilità: ridefinire un metodo

- Una sottoclasse può anche ridefinire un metodo

```
public class Corona extends Circle{
    private double raggiointerno;

    public Corona(double r1, r2) {/*code omitted */}

    public void area() {/* codice per l'area della corona */}
}

Corona dc = new Corona(3, 5);

double a = dc.area();        // area della corona !

Circle c = dc;    // c è dc "visto come" un Circle
```



S.Martini, Linguaggi orientati agli oggetti

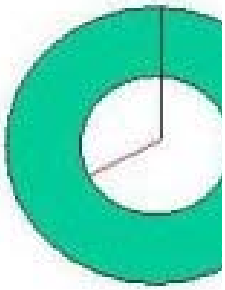
L'esempio del tempo con i secondi lo potete guardare in dettaglio da soli, più interessante è discutere un altro concetto che è la ridefinizione di un metodo, che garantisce l'estensibilità delle classi. Fino adesso abbiamo definito una sottoclasse per sola estensione, aggiungendo informazione, in questa aggiunta una sottoclasse può anche ridefinire un metodo. Vediamo questo esempio in cui definiamo una classe Corona (intendiamo la corona circolare) che estende la classe circle. Una corona è costituita da due cerchi concentrici, aggiungiamo quindi un altro parametro che sarà il raggio interno.

Estensibilità: ridefinire un metodo

Estensibilità: ridefinire un metodo

- Una sottoclasse può anche ridefinire un metodo

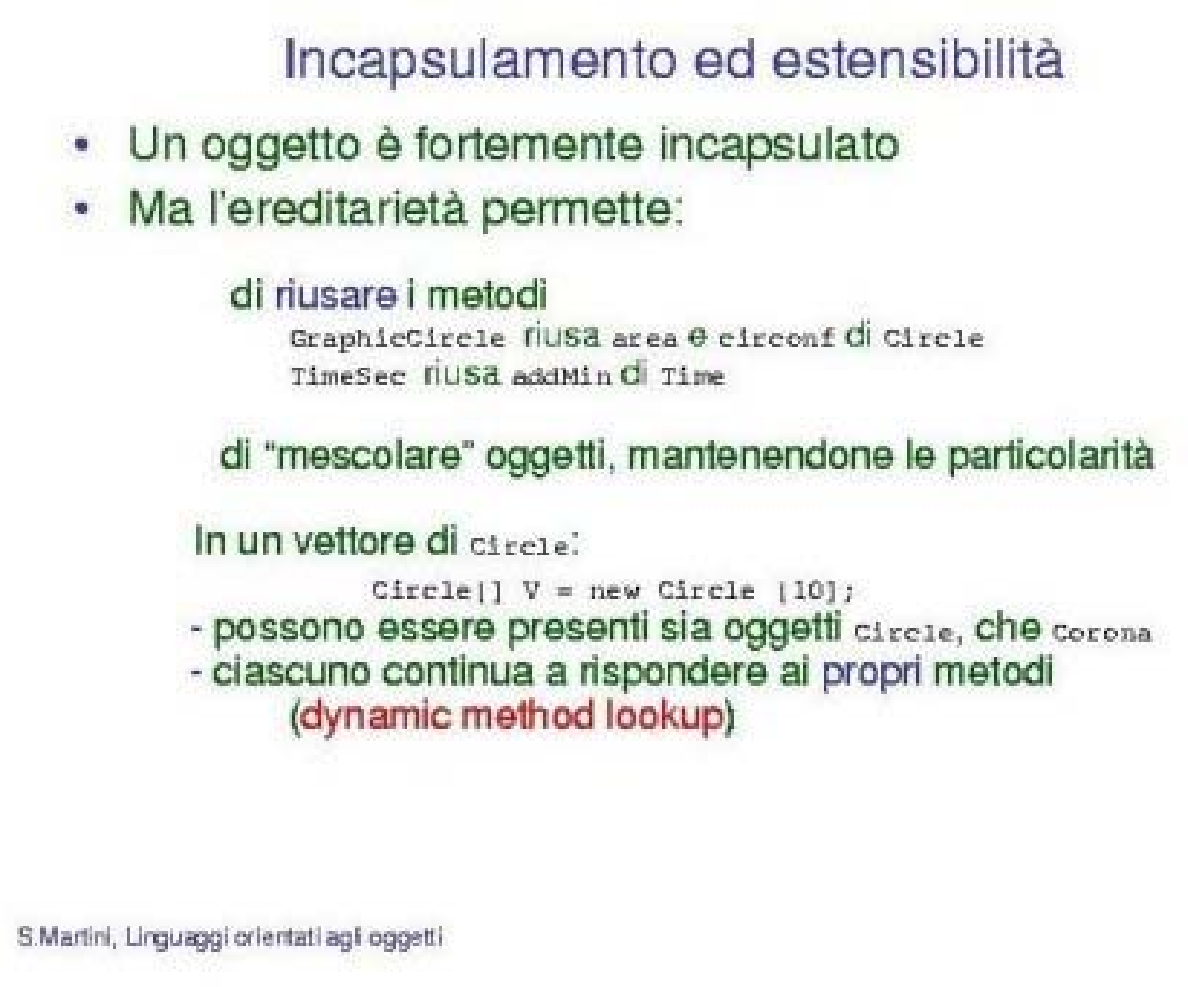
```
public class Corona extends Circle{
    private double raggiointerno;

    public Corona(double r1, r2) {/*code omitted */}

    public void area() {/* codice per l'area della corona */}
}

Corona dc = new Corona(3, 5);

double a = dc.area();        // area della corona !

Circle c = dc;    // c è dc "visto come" un Circle
```



S.Martini, Linguaggi orientati agli oggetti

Abbiamo il costruttore, chiamato anch'esso Corona, che avrà in ingresso due raggi: il primo interno r1 e il secondo esterno r2. Abbiamo poi il metodo area che ci restituisce l'area della corona. Questo metodo era anche definito nel circle, in tal caso lo stiamo ridefinendo. Ogni linguaggio orientato agli oggetti permette la ridefinizione dei metodi. In questo modo gli oggetti della nuova sottoclasse corona rispondono al metodo area definito dentro la sottoclasse e non quello definito nella sopraclasse. Riprendendo l'esempio abbiamo la costruzione di una nuova corona dc con raggio interno 3 e raggio esterno 5; dc è un oggetto della sottoclasse Corona della classe circle e quindi risponde al metodo area, ma il metodo invocato non è quello della sopraclasse circle, ma il metodo della sottoclasse Corona, cioè viene calcolata l'area della corona.

Incapsulamento ed estensibilità



Concludiamo con un po' di osservazioni topologiche. Un oggetto è fortemente incapsulato, ma l'ereditarietà, cioé la possibilità di definire sottoclassi, di ereditare i metodi della sopraclasse nella sottoclasse, consente di riusare i metodi: GraphicCircle riusa area e circonf che erano definiti in circle; TimeSec riusa addMin che era definito in Time. Oltre a consentire il riuso, viene ammessa anche la ridefinizione, riuscendo a mescolare oggetti mantenendone le particolarità. Ad esempio possiamo costruire un vettore di cerchi, chiamato cv, dove possono esserci oggetti che siano sia cerchi che corone. Una cosa importante su cui possiamo insistere poco, perché è un concetto abbastanza avanzato, è che, nonostante facciano tutti parte di un vettore definito come circle, ciascuno di questi oggetti risponderà però ai propri metodi: un oggetto cerchio risponde ad Area come area del cerchio, una corona risponderà invece con l'area della corona. Questa proprietà molto importante viene chiamata nel gergo orientato agli oggetti look up dinamico dei metodi (ricerca dinamica dei metodi).

Dynamic method lookup

Dynamic method lookup

- Immaginiamo che in `TimeSec` che estende `Time`
  `print()` di `Time` sarà ridefinito in `TimeSec`
  per stampare anche i secondi

  In un vettore di oggetti `Time`, alcuni possono essere `TimeSec`

```
Time[] time_vect = new Time[10];

time_vect[1] = Time(20,30);
time_vect[5] = TimeSec(21,30,14);

for(int i=0; i<10; i++)
      time_vect[i].print();
```

  Ogni oggetto usa il proprio metodo `print` !
- La selezione avviene in modo dinamico
  durante l'esecuzione

S.Martini, Linguaggi orientati agli oggetti 26

Su questa trasparenza insiste su questo argomento, come al solito i dettagli li potete vedere più comodamente da soli, anche perché riprende le classi TimeSec e Time. Immaginiamo che dentro queste classi siano stati inseriti anche dei metodi Print, naturalmente in Time verranno stampati solo ore e minuti, in TimeSec ore, minuti e secondi. Preso un vettore di oggetti Time, alcuni dei suoi elementi possono essere TimeSec; nel caso volessimo stampare quel vettore, vorremo che i Time vengano stampati come Time e i TimeSec come TimeSec. Ogni oggetto userà automaticamente il proprio metodo e la selezione avviene in modo dinamico durante di esecuzione, ecco perché viene chiamata ricerca dinamica dei metodi, questa infatti non può avvenire a tempo di compilazione, ma durante l'esecuzione.

Conclusioni

Conclusioni

Classi e sottoclassi
Oggetti e ereditarietà

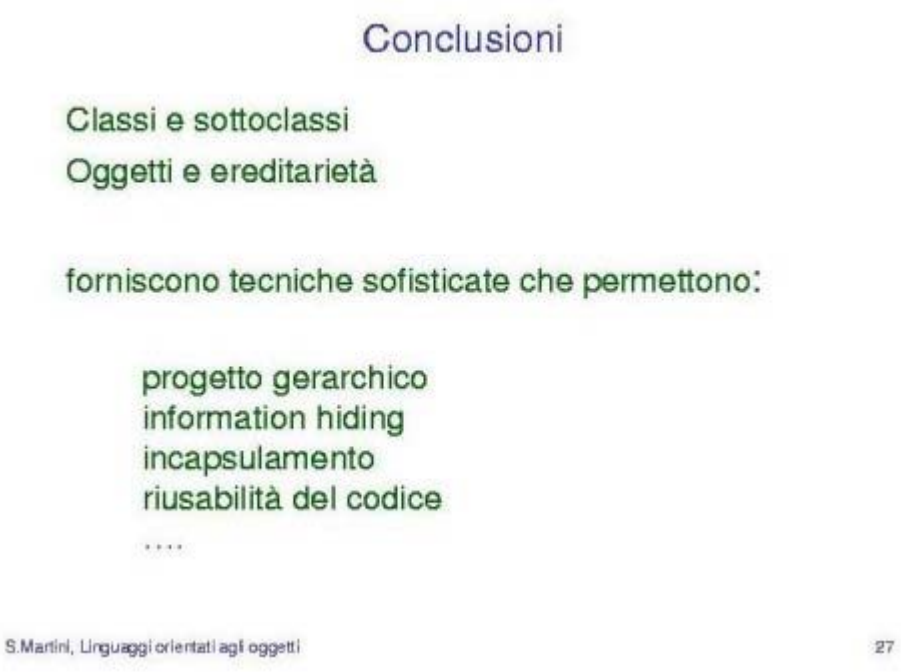
forniscono tecniche sofisticate che permettono:

progetto gerarchico
information hiding
incapsulamento
riusabilità del codice
....

S.Martini, Linguaggi orientati agli oggetti 27

In conclusione i linguaggi orientati agli oggetti, Java in particolare, sono uno strumento molto flessibile che permette non solo la costruzione di programmi complessi, ma sono uno strumento metodologico che insegna a strutturare soluzioni e, in questo ruolo sono utili anche al di fuori dell'informatica, intesi come una tecnologia per costruire sistemi complessi.

Conclusioni



Possiamo concludere questa lezione con un po' di osservazioni metodologiche. Abbiamo visto come i concetti fondamentali della programmazione object oriented siano quelli di classe e di sottoclasse, cioé la possibilità di definire gerarchie di classe e quindi di avere oggetti che ereditano proprietà da classi superiori. I costrutti linguistici forniscono delle tecniche molto sofisticate che permettono di raggiungere all'interno di un progetto quelle caratteristiche che avevamo visto essere molto importanti per la progettazione di sistemi complessi. In un progetto gerarchico, le classi permettono di definire gerarchie all'utente, consentono il nascondimento dell'informazione, sia perché le classi stesse sono un meccanismo di incapsulamento, sia perché si possono utilizzare i due parametri private o public in modo tale da garantire una totale o una parziale information hiding. Un progetto gerarchico consente inoltre di riutilizzare il codice già scritto in modo pulito, senza doverlo stanziare ogni volta in contesti diversi; naturalmente qui gioca un ruolo essenziale il concetto di ereditarietà del linguaggio. Potremmo elencare ancora tante altre qualità che non abbiamo visto.