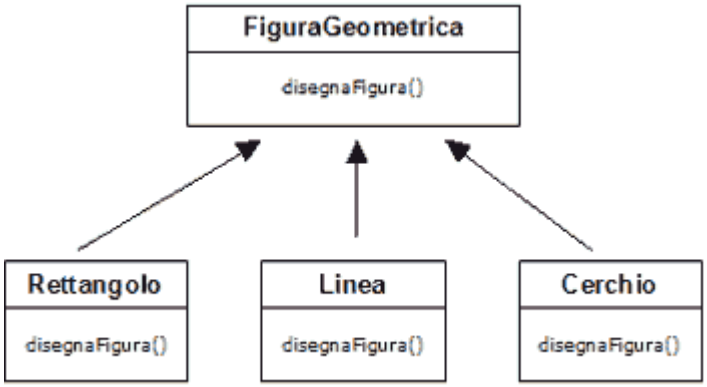
POLIFORMISMO

I

l terzo elemento fondamentale della programmazione ad Oggetti è il polimorfismo. Per rendere l'idea più chiara, utilizzando ancora una volta un esempio del mondo reale, si pensi al diverso comportamento che assumono un uomo, una scimmia e un canguro quando eseguono l'azione del camminare. L'uomo camminerà in modo eretto, la scimmia in maniera decisamente più goffa e curva mentre il canguro interpreterà tale azione saltellando.

Riferendoci ad un sistema software ad oggetti, il polimorfismo indicherà l'attitudine di un oggetto a mostrare più implementazioni per una singola funzionalità. Per esempio, supponiamo di voler costruire un sistema software in grado di disegnare delle figure geometriche. Per tale sistema avremo definito una classe padre chiamata FiguraGeometrica dalla quale avremo fatto derivare tutte le classi che si occupano della gestione di una figura geometrica ben precisa. Per maggiore chiarezza riportiamo quanto detto nel diagramma seguente:

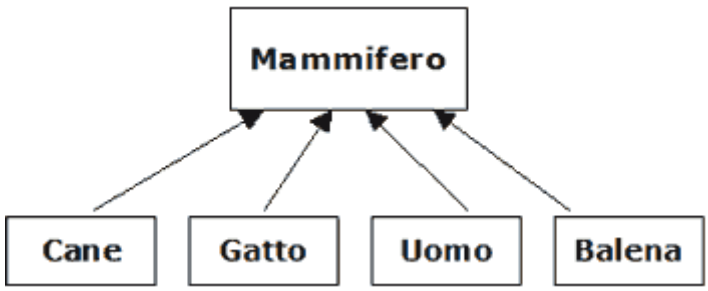


Quando l'utente desidera rappresentare una di tali figure, sia essa una linea, un cerchio o un rettangolo, egli eseguirà una determinata azione che produrrà l'invio al sistema di un messaggio che, a sua volta, scatenerà l'invocazione del metodo disegnaFigura della classe FiguraGeometrica..

Con l'utilizzo del polimorfismo, il sistema è in grado di capire autonomamente quale figura geometrica debba essere disegnata ed invocarne direttamente il metodo disegnaFigura appartenente alla classe figlia coinvolta.

L’ereditarietà

L’ereditarietà costituisce il secondo principio fondamentale della programmazione ad oggetti. In generale, essa rappresenta un meccanismo che consente di creare nuovi oggetti che siano basati su altri già definiti. Si definisce **oggetto figlio** (child object) quello che eredita tutte o parte delle proprietà e dei metodi definiti nell’**oggetto padre** (parent object).È semplice poter osservare esempi di ereditarietà nel mondo reale. Ad esempio, esistono al mondo centinaia tipologie diverse di mammiferi: cani, gatti, uomini, balene e così via. Ognuna di tali tipologie di mammiferi possiede alcune caratteristiche che sono strettamente proprie (ad esempio, soltanto l’uomo è in grado di parlare) mentre esistono, d’altra parte, determinate caratteristiche che sono comuni a tutti i mammiferi (ad esempio, tutti i mammiferi hanno il sangue caldo e nutrono i loro piccoli). Nel mondo Object Oriented, potremmo riportare tale esempio definendo un oggetto Mammifero che inglobi tutte le caratteristiche comuni ad ogni mammifero. Da esso, poi, deriverebbero gli altri child object: Cane, Gatto, Uomo, Balena, etc. L’oggetto cane, per citarne uno, erediterà, quindi, tutte le caratteristiche dell’oggetto mammifero e a sua volta conterrà delle caratteristiche aggiuntive, distintive di tutti i cani come ad esempio: ringhiare o abbaiare. Il paradigma OOP, ha quindi carpito l’idea dell’ereditarietà dal mondo reale, come mostrato nella figura seguente:



e, pertanto lo stesso concetto viene applicato ai sistemi software che utilizzano tale tecnologia.

Uno dei maggiori vantaggi derivanti dall’uso dell’ereditarietà è la maggiore facilità nella manutenzione del software. Infatti, rifacendoci all’esempio dei mammiferi, se qualcosa dovesse variare per l’intera classe dei mammiferi, sarà sufficiente modificare soltanto l’oggetto padre per consentire che tutti gli oggetti figli ereditino la nuova caratteristica.

Ad esempio, se i mammiferi diventassero improvvisamente (e anche inverosimilmente!) degli animali a sangue freddo, soltanto l’oggetto padre mammifero necessiterà di tale variazione. Il gatto il cane, l’uomo, la balena, e tutti gli altri oggetti figli erediterebbero automaticamente la caratteristica di avere il sangue freddo, senza nessuna modifica.

Un esempio che metta in luce la potenza dell’ereditarietà, in un programma Windows object oriented potrebbe avere come oggetto fulcro le finestre associate al programma stesso. Supponiamo,

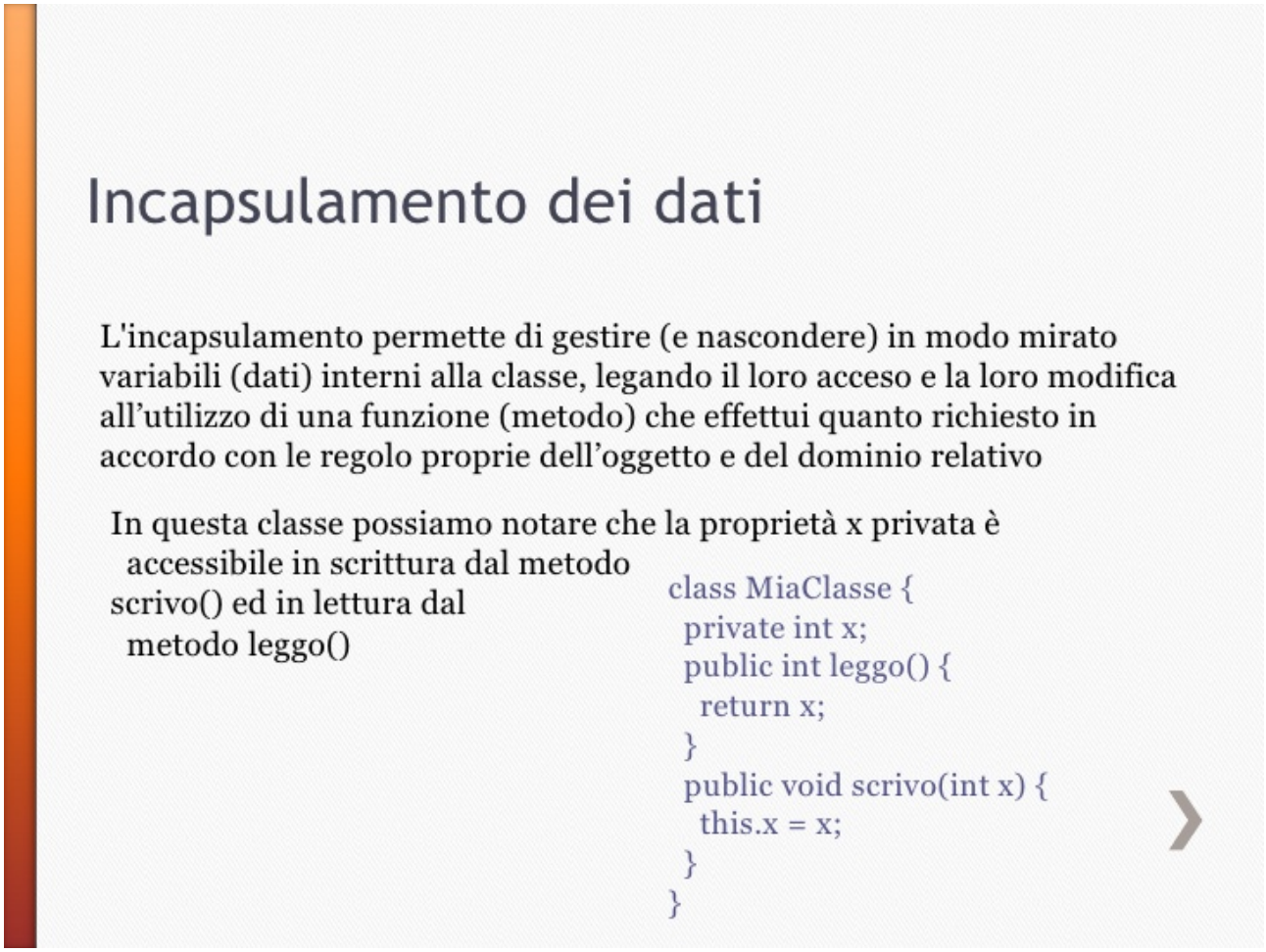
ad esempio, che tale software utilizzi, in totale, 100 finestre differenti, visualizzabili a secondo del contesto in cui sta navigando l'utente.

INCAPSULAMENTO

L'incapsulamento rappresenta il principio in base al quale una classe può mascherare la sua struttura interna e proibire ad altri oggetti di accedere ai suoi dati o di richiamare le sue funzioni che non siano direttamente accessibili dall'esterno.

Lo scopo principale dell'incapsulamento è di dare accesso allo stato e ai comportamenti di un oggetto solo attraverso un sottoinsieme di elementi pubblici.

L'incapsulamento permette quindi di considerare una classe come una sorta di "scatola nera", che permette di mostrare solo ciò che è necessario, mascherando ciò che non deve trasparire verso l'esterno.



Persistenza

In informatica, il concetto di persistenza si riferisce alla caratteristica dei dati di sopravvivere all'esecuzione del programma che li ha creati: senza questa capacità infatti i dati vengono salvati solo in memoria Ram e verranno persi allo spegnimento del computer.

Nella programmazione informatica, la persistenza si riferisce in particolare alla possibilità di far sopravvivere delle strutture dati all'esecuzione di un singolo programma. Questa possibilità è raggiunta salvando i dati in uno storage non volatile, come su un file system o su un database (es. applicazioni web).

Esempi di persistenza sono usati ad esempio nella serializzazione Java per salvare oggetti Java su disco rigido, oppure in Java EE per salvare i dati di un Enterprise Java Beans in un database nell'ambito di applicazioni web in architettura three-tier.