APPUNTI PER IL CORSO DI PROGRAMMAZIONE (SEI 1° MODULO)

ANNO 2002/2003

PAOLA CAMPADELLI

Note sul concetto di Linguaggio

Linguaggio e Realtà

Un linguaggio serve a parlare di una certa realtà.
Per ragionare su un linguaggio bisogna considerare anche la realtà di cui il linguaggio parla (deve parlare): **l'universo del discorso** del linguaggio.
Bisogna considerare due mondi distinti ma collegati.

1. il mondo della **realtà** di cui il linguaggio parla (case, alberi, persone, numeri, poligoni, ...): oggetti **reali**; sitazioni reali; cose; stati di cose; concetti ...
2. il mondo del **linguaggio** (discorsi, frasi, espressioni, parole, ...): oggetti **lessicali**.

Tra i due mondi, se il linguaggio ha un significato, c'è una relazione: la **relazione semantica**, a volte chiamata anche **funzione semantica**:

funzione semantica: elementi del linguaggio → elementi della realtà.

Si dice che un oggetto lessicale (elemento del linguaggio) **denota** (esprime) un oggetto reale o uno stato di cose reale (elemento della realtà).

Strutture

Generalmente nel mondo di cui si vuole parlare è riconoscibile una molteplicità di oggetti distinti.
Questi oggetti sono tra loro collegati, in relazione; si compongono in **strutture**, che a loro volta si compongono in strutture più complesse. Non c'è limite ai livelli di complessità che nel mondo si possono incontrare.
I mondi di cui vogliamo parlare sono mondi complessi e strutturati.
Il linguaggio usato per parlare imita la struttura del mondo. A oggetti reali semplici corrispondono oggetti lessicali semplici. A strutture reali complesse corrispondono strutture lessicali complesse. Generalmente i linguaggi non pongono limiti alla complessità delle proprie strutture.
Esistono convenzioni, regole per la composizione/decomposizione delle strutture linguistiche. Sono le regole della **sintassi** del linguaggio.

Serializzazione

Le strutture del linguaggio, per quanto complesse, hanno una caratteristica fondamentale: esse posono esere serializzate, cioè possono essere rese mediante sequenze di simboli (oggetti lessicali elementari). Strutture logicamente anche molto complicate, dotate di molti livelli logici, possono essere "appiattite" su un unico livello sequenziale inserendo particolari oggetti lessicali (congiunzioni, connettivi, parentesi, avverbi, segni di interpunzione ...). Ciò perchè il linguaggio nasce per consentire all' uomo di esprimere situazioni attraverso sequenze di suoni (fonemi). I fonemi si possono pronunciare (ascoltare) solamente in sequenza.
Strettamente collegate alla natura del linguaggio sono quindi due operazioni:

1. La traduzione di una struttura linguistica complessa in una sequenza di oggetti lessicali semplici (parole, simboli). (**Serializzazione**).
2. L'operazione inversa: la ricostruzione di una struttura complessa a partire da una sequenza di simboli. (**Analisi sintattica** o **Parsing**).

Espressioni di un linguaggio

Un discorso di un certo linguaggio può essere suddiviso in **espressioni**. Un'espressione è un frammento di discorso. Periodi, frasi, parole sono espressioni. Un pezzo di frase è un'espressione.
Tra le espressioni, alcune sono particolarmente notevoli, e meritano un nome:

- una **frase** è un'espressione dotata di "senso compiuto"; denota uno stato di cose dell'universo del discorso.
- un **termine** è un'espressione che denota un oggetto reale.
- un **connettivo** è un'espressione che si usa per connettere altre espressioni.
- un **predicato** è un'espressione che denota una relazione.

Esempi:

- "Romolo è il primo re di Roma" è una frase.
- "Romolo" è un termine. Denota una persona.
- "il primo re di Roma" è un termine. Denota una persona.
- "Roma" è un termine. Denota una città.

- "è" è un predicato.
- "2+3=5 & 5 è dispari" è una frase. Denota uno stato di cose nel mondo dell'aritmetica.
- "2+3=5" è una frase; "5 è dispari" è un'altra frase;
- "&" è un connettivo.
- "è dispari" è un predicato.
- "2", "3", "5" sono termini; denotano tre numeri ben conosciuti.
- "2+3" è un termine; denota il numero che si ottiene sommando il numero 2 al numero 3, cioè il numero 5.
- "=" è un predicato.

Termini

Tradizionalmente si distinguono tre tipi di termini:

1. **Costanti** o nomi propri. Sono espressioni che, quando usate, denotano sempre lo stesso oggetto. Esempi: "Romolo", "Roma", "2", "3", "π".
2. **Termini funzionali**:
   - "Il primo re di Roma"
   - "sin(90°)"
   - "2+3"

   Schematicamente un termine funzionale ha la forma f(t1, ..., tn) dove f è un simbolo di funzione e t1, t2, ..., tn sono altri termini qualsiasi:
   - primoReDi(Roma)
   - sin(90°)
   - +(2, 3)

   Non sempre la forma schematica presentata (notazione prefissa) è la preferita.
3. **Variabili.** vedi di seguito.

Variabili

L'uso di costanti e di termini funzionali che contengono solo costanti consente di esprimere molto bene fatti particolari, ma non fatti generali.
Per questo nei linguaggi si introducono le **variabili**.
Una variabile è un nome che non denota nessun oggetto in particolare, ma può (potrebbe) denotarli tutti.
Consideriamo l'espressione E = "X è il primo re di Roma". Essa contiene la variabile X.
L'espressione E assomiglia a una frase, ha la stessa forma di una frase, ma non è dotata di senso compiuto perchè il termine X non denota alcun oggetto particolare. Essa è uno "schema di frase". Può diventare una frase mediante un **assegnamento** della variabile X.
L'assegnamento è l'operazione in cui si stabilisce che una variabile denota un certo oggetto reale.
Prima dell'assegnamento la variabile non denota.
Dopo dell'assegnamento la variabile denota l'oggetto assegnato.
Esempio:

Sia X = Romolo
X è il primo re di Roma

L'assegnamento ha prodotto una frase vera. Un altro assegnamento può produrre una frase falsa.

Sia X = Giulio Cesare
X è il primo re di Roma.

Il concetto di assegnamento richiama il concetto di **contesto**. L'assegnamento apre un contesto in cui la variabile denota l'oggetto.

Quantificazione di variabili

L'assegnamento non è l'unica operazione che fornisce un senso compiuto ad una frase contenente variabili. A volte (spesso) si formano frasi come

- Per ogni X (X è mortale)
- Per qualche Y (Y è il primo re di Roma)

La prima frase dice che **per ogni assegnamento di X** X è mortale; la seconda dice che **per qualche assegnamento** (almeno uno) Y è il primo re di Roma.

Tipi

In alcuni mondi esistono i Tipi, cioè gli oggetti reali possono essere raggruppati in classi dotate di caratteristiche comuni. Si tratta di mondi particolarmente "regolari".
Esempio di tipi:

- il tipo NumeroIntero
- il tipo EssereUmano
- il tipo Città.

In un mondo dotato di tipi generalmente esiste un ordinamento parziale tra i tipi (relazione di **sottotipo**)

- NumeroIntero ≤ Numero
- EssereUmano ≤ Mammifero ≤ Animale

L'ordinamento è parziale: né Animale ≤ Numero, né Numero ≤ Animale.

Linguaggi con tipi

Per parlare di un mondo dotato di tipi è utile un linguaggio dotato di tipi. A ogni **tipo** (nell'universo del discorso) viene assegnato un **nome di tipo** nel linguaggio.
Se è vero che ogni oggetto reale appartiene ad (almeno) un tipo, anche ogni termine (oggetto lessicale che denota un oggetto reale) dovrà essere associato a un tipo (attraverso il nome del tipo).
Ciò non presenta alcuna difficoltà per i nomi propri: essi nascono associati ad un particolare oggetto e ne assumono il tipo.

- "Romolo" denota Romolo, che è una Persona; quindi "Romolo" è un termine di tipo Persona.
- "2" denota 2, che è un numero; quindi "2" è un termine di tipo Numero.

Le cose non sono altrettanto semplici per variabili e termini funzionali.

Tipo di una variabile

Qual è il tipo della variabile X nella frase "Ogni X è pari o dispari"?
Non c'è modo di saperlo, se non lo si dichiara.
E' necessario introdurre, nei linguaggi dotati di tipi, una operazione aggiuntiva per le variabili: **la dichiarazione di tipo**.
Essa può essere a sé stante oppure incorporata nella frase che contiene x.

1. Sia X di tipo Persona. Ogni X è mortale.
2. Ogni X di tipo Persona è mortale.
3. Per ogni X:Intero (dispari(X) | pari(X))

Se il linguaggio è "fortemente tipato", cioè particolarmente rigoroso a proposito di tipi, non sarà in esso possibile (lecito) utilizzare una variabile di cui non sia stato dichiarato il tipo.
Anche la dichiarazione di tipo ha un contesto di validità.

Tipo di una funzione

Analogo discorso vale per le funzioni. Per esse bisognerà dichiarare il tipo degli argomenti e del risultato:

- sin: NumeroReale → NumeroReale
- il primo re di: Città → Persona
- la data di matrimonio di: Uomo, Donna → Data

Tipo di un termine

A questo punto ogni termine di un linguaggio tipato è dotato di tipo.

- "Romolo" è un termine di tipo Persona perchè è una costante di tipo Persona
- "Roma" è un termine di tipo Città perchè è una costante di tipo Città.
- "il primo re di Roma" è un termine di tipo Persona perchè deriva dalla applicazione della funzione
  il primo re di: Città → Persona
  alla città di Roma.
- "il primo re di Giulio Cesare" è un'espressione senza senso, perchè la funzione precedente è applicata ad un oggetto di un tipo non consentito.
- Nella frase "Per ogni x di tipo Persona, x è mortale" la variabile x è di tipo Persona perché così dichiarato esplicitamente.

Nota. I linguaggi naturali massimizzano l'efficienza nell'uso delle parole. In essi si dirà probabilmente "Ogni persona è mortale". In questa espressione il nome proprio del tipo viene tramutato in nome comune di oggetto, e assolve al ruolo di variabile tipata.

Azioni e linguaggi procedurali

Tutti i linguaggi devono denotare stati di cose. Alcuni linguaggi vogliono denotare anche **azioni** che trasformano lo stato delle cose. Azioni semplici possono essere composte producendo azioni composte.
Alcune modalità di composizione delle azioni: **sequenza**, **iterazione**, **alternativa**, ...
Analogamente il linguaggio avrà **espressioni** che denotano azioni semplici, e espressioni che denotano sequenze, iterazioni, alternative. Alcune parole del linguaggio saranno dedicate a denotare le azioni semplici; altre saranno riservate a significare le operazioni di composizione tra azioni.

Azioni e tipi

Un linguaggio che parla di azioni e comprende la nozione di tipo distingue, per ciascuna azione (tipo di azione), il tipo degli oggetti a cui essa può essere applicata, e (se ce ne sono) il tipo degli oggetti da essa prodotta. Vale un discorso analogo a quanto detto prima sulle funzioni e i termini funzionali.

Linguaggi orientati agli oggetti

Esistono visioni del mondo in cui ciascun oggetto è dotato di certe capacità di azione, e tutte le azioni sono azioni che competono a, sono responsabilità di, qualche oggetto. Se ci sono i tipi, il discorso si trasferisce ai tipi. Anzi, le specifiche capacità di azione sono caratteristica essenziale di un tipo. Un tipo è un insieme strutturato di capacità di azione, e quindi di interazione con altri oggetti.
Un oggetto può eseguire tutte le azioni di sua competenza. Nessun oggetto può eseguire azioni non di sua competenza. Un oggetto A può ottenere il risultato di un'azione di competenza di B solo **invocando** l'esecuzione dell'azione da parte di chi ne è capace (B). Il funzionamento generale del mondo nasce dalla cooperazione dei diversi oggetti che lo compongono.

Il linguaggio Java

Un linguaggio formale, object oriented, fortemente tipato. Parla di oggetti primitivi esistenti nel calcolatore (esenzialmente numeri interi) e di oggetti composti che si ottengono per composizione di oggetti semplici.
E' un linguaggio procedurale, e descrive compiti anche molto complicati attraverso l'interazione di oggetti capaci di azioni.

Definizione di un Linguaggio Formale

Un linguaggio formale, qual è un linguaggio di programmazione, deve essere definito in modo chiaro, preciso e completo. Una specifica chiara precisa e completa è necessaria sia per chi deve costruire compilatori e/o interpreti del linguaggio, sia per chi deve scrivere programmi. Cosa vuol dire descrivere con precisione un linguaggio? Definirne con precisione la sintassi e la semantica.
La **sintassi** specifica sia la struttura lessicale del linguaggio (come costruire le parole a partire dai caratteri dell'alfabeto) sia le regole per la costruzione delle frasi ben formate (programmi sintatticamente corretti).
La **semantica** specifica il significato delle frasi (programmi). Specificare in modo formale la sintassi è ragionevolmente semplice, specificarne la semantica non lo è; ci sono diversi metodi per precisare in modo formale la semantica: operazionale (fare corrispondere a frasi del linguaggio operazioni eseguite dalla macchina), denotazionale, assiomatico. Noi non analizzeremo questi metodi e descriveremo il linguaggio Java come si fa usualmente mediante una combinazione di sintassi formale e di semantica informale, cioè specificata usando il linguaggio naturale. Purrtroppo ciò, in qualche caso, può dar luogo ad interpretazioni conflittuali.

Sintassi

La sintassi di un linguaggio è l'insieme di regole che governano la formazione delle espressioni significative (**discorsi**, **frasi**) a partire da un insieme di costituenti elementari (**parole** di un certo **dizionario**, oppure, se si vuole trattare anche la composizione delle singole parole, **caratteri** di un **alfabeto**).
Si tratta di regole di **composizione**: componendo le lettere dell'alfabeto si ottengono le parole; componendo le parole si ottengono espressioni; componendo espressioni si ottengono espressioni più complesse; e così via.
Le strutture (espressioni) ottenibili in un linguaggio sono generalmente di complessità illimitata, in quanto le regole di composizione si possono applicare un numero illimitato di volte. Tuttavia per fortuna le modalità di composizione sono in generale piuttosto semplici. La operazione di composizione che sta alla base di tutto è la semplice **giustapposizione** di elementi.

Regole sintattiche

Una regola sintattica dice come si compone una espressione di tipo A a partire da altre espressioni di tipo B, C, D etc.
Esempio:

```
(1) Frase := Soggetto Predicato
```

La regola (1) dice che un modo di comporre una frase è quello di giustapporre un soggetto con un predicato.
La regola (1) non dice che cos'è (come si può comporre) un soggetto; nè dice che cos'è un predicato. Quindi la sola regola (1) non è sufficiente a definire la sintassi di un linguaggio, per quanto esso possa essere semplice.
Proviamo a completarla:

```
(2) Soggetto := FormaNominale
(3) FormaNominale := NomeProprio
(4) FormaNominale := Articolo NomeComune
(5) Predicato := FormaVerbaleIntransitiva
(6) Predicato := FormaVerbaleTransitiva FormaNominale
(7) Articolo := "il" |  "un"
(8) NomeProprio := "Carlo" | "Maria"
(9) NomeComune := "cane" | "gatto"
(10) FormaVerbaleIntransitiva := "passeggia"
(11) FormaVerbaleTransitiva := "rincorre"
```

L'insieme delle regole (1,…,11) consente di produrre le frasi:

- Il cane rincorre un gatto
- Maria passeggia
- Carlo rincorre Maria

ed altre quali Maria rincorre Carlo, un gatto rincorre il cane,…ma non le frasi

- Oggi piove
- cane passeggia gatto

Delle ultime due espressioni la prima utilizza parole sconosciute; la seconda compone parole conosciute con modalità non in accordo con le regole della sintassi.
Mostriamo come la frase 'Maria Passeggia' possa essere derivata usando le regole sopra descritte mediante una successione di sostituzioni effettuate a partire dalla parola Frase. Per sostituzione o derivazione intendiamo che se A := B è una delle regole sopra elencate, allora ogni volta che A si presenta come sottoparola della parola A' può essere sostituito da B.

```
Frase → Soggetto Predicato  (regola 1)
Soggetto Predicato → FormaNominale Predicato (regola 2)
FormaNominale Predicato → NomeProprio Predicato (regola 3)
NomeProprio Predicato → "Maria" Predicato (regola 8)
"Maria" Predicato → "Maria" "passeggia" (regola 10)
```

Osserviamo le regole sopra descritte. Ciascuna regola ha la forma

FormaDaDefinire := *Definizione*

FormaDaDefinire è il nome di una espressione sintattica.
Definizione è una espressione formata da

- nomi di espressioni sintattiche (come *Articolo*, *NomeProprio*, ...)
- vocaboli del linguaggio oggetto di definizione (espressi tra virgolette; come "*Carlo*", "*Maria*", "*gatto*", ...);
- operatori di composizione (come |)

Ci chiediamo: in quale linguaggio è scritta la definizione (1,...,11)?
Non certo nel linguaggio che stiamo definendo (linguaggio oggetto): infatti il linguaggio da definire parla di cani e gatti, e invece le nostre regole parlano di articoli e forme sintattiche.
Si tratta di un linguaggio speciale costruito per parlare di altri linguaggi, un cosiddetto **metalinguaggio**.
Esso contiene parole sue proprie, che non appartengono al linguaggio oggetto (come NomeProprio, Articolo, ...); ma esso ha bisogno anche di denotare le parole del linguaggio oggetto ("Maria", "cane", "rincorre", ...).
Si tratta di un (meta)linguaggio formale detto **notazione BNF** (Backus Naur Form).

- Ogni regola si dice regola di **produzione**
- Le parole utilizzate per denotare parole del linguaggio oggetto sono dette **simboli terminali**.
- Le parole utilizzate per denotare forme sintattiche (tipi di espressioni) sono dette **simboli non terminali**.

Questa terminologia può apparire curiosa; essa diventa comprensibile se si pensa alle regole come stampi che **producono** espressioni; quando si incontra il nome di un altro stampo bisogna andare avanti (*non terminale*); quando si incontra il nome di una parola del linguaggio oggetto ci si può fermare (*terminale*).

Il simbolo "|" denota la regola di composizione **alternativa,** A:=B|C è pertanto una abbreviazione per le due produzioni A:=B, A:=C;
Altre operazioni possibili sono

- la ripetizione, spesso denotata con {...};
- la presenza eventuale, spesso denotata con [...].Quindi [A] denota "forse A". Più precisamente possiamo dire che `A:=B[C]D abbrevia A:=BCD|BD.`

Consideriamo due esempi più rigorosi: usiamo la notazione BNF per generare stringhe che rappresentano numeri con decimali e per generare stringhe che, secondo la usuale notazione matematica, rappresentano espressioni aritmetiche. Rappresentiamo un numero con decimali, quale ad esempio 5.2, mediante una parte intera, un punto, una parte frazionaria. La parte intera può non essere presente (es .5) o essere costituita da una sequenza di digit. La parte frazionaria è costituita da una sequenza di digit. Una sequenza di digit è un digit oppure un digit seguito da una sequenza di digit. Infine un digit è una delle dieci cifre "0"..."9".
I simboli terminali sono le 10 cifre, come metasimboli (parole del metalinguaggio) utilizziamo Numero, ParteIntera, ParteFrazionaria, Digit, DigitSequence. Mediante la BNF possiamo scrivere le produzioni:

```
Numero := [ParteIntera] "." ParteFrazionaria
ParteIntera := DigitSequence
ParteFrazionaria := DigitSequence
DigitSequence := Digit | Digit DigitSequence
Digit := "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9"
```

Anche in questo caso ci troviamo di fronte ai due costrutti, la sequenza e la scelta, che abbiamo incontrato nell'esempio precedente. Osserviamo inoltre che la regola che definisce la DigitSequence è ricorsiva (DigitSequence è definito in termini di un Digit e di una DigitSequence più corta).
Un modo alternativo di definire la DigitSequence è `DigitSequence := {Digit}`.
Nei linguaggi di programmazione ci troveremo a dover definire costrutti che rappresentano una sequenza, una scelta, oppure una iterazione.

Per i numeri con decimali, ci sono rappresentazioni alternative a quella che abbiamo utilizzato; per esempio il numero 3.14 potrebbe essere scritto nei seguenti modi: `314*E-2; 0.314E+1; 0.314*E1; .314E1` (dove E-2,E1, E+1 stanno rispettivamente per 10^-2 e 10^1 e *, dove presente, indica il prodotto.

Consideriamo ora una BNF per espressioni numeriche con gli operatori + , -,*, / e con le parentesi rotonde per raggruppare sottoespressioni.

```
EsprNum := Somma
Somma := Prodotto [PiuMeno Somma]
Prodotto := Fattore [PerDiviso Prodotto]
Fattore := "(" EsprNum ")" | Numero
Numero := [ParteIntera] "." ParteFrazionaria
ParteIntera := {Digit}
ParteFrazionaria := {Digit}
Digit := "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9"
PiuMeno := "+"|"-"
PerDiviso := "*"|"/"
```

La BNF è la notazione più comunemente usata per rappresentare l'insieme di regole, **grammatica generativa**, che descrive con precisione il lessico ed i costrutti di un linguaggio di programmazione. Lo studio delle grammatiche formali (generative) ha un'origine storica nei lavori di Chomsky che ne ha proposto una classificazione basata sulla loro adeguatezza a descrivere diversi aspetti del fenomeno linguistico e sulla possibilità di una trattazione matematica. Per i nostri scopi ci limitiamo a considerare grammatiche in cui le regole di produzione specificano come un metasimbolo possa essere sostituito da una sequenza di simboli terminali e metasimboli. Tali grammatiche sono dette **grammatiche non contestuali** (context-free).

Sintassi di Java

La struttura lessicale e la struttura sintattica di un programma Java sono definite mediante grammatiche non contestuali chiamate rispettivamente *lexical grammar* and *syntactic grammar*.
La *lexical grammar* ha come insieme di simboli terminali un alfabeto particolarmente ricco l' Unicode; essa descrive come una sequenza di caratteri Unicode è trasformata in una sequenza di elementi di input.
La *syntactic grammar* descrive come una sequenza di elementi lessicali significativi (tokens) devono essere composti per costruire programmi sintatticamente corretti.

Lexical grammar

L'alfabeto del linguaggio Java è l'Unicode, un insieme di 65536 caratteri che è stato definito internazionalmente (http//www.unicode.org) per tenere conto degli alfabeti di tutte le lingue naturali. I caratteri dell'alfabeto sono posti in corrispondenza biunivoca con i naturali {0, …, 65535} ed ogni carattere è rappresentato da 2 byte.
I primi 128 caratteri Unicode sono i caratteri ASCII (American Standard Code for Information Interchange); essi possono essere utilizzati per rappresentare qualsiasi carattere Unicode mediante la sequenza costituita da `\u` seguito da 4 digit esadecimali che denotano l'intero corrispondente:

```
Unicode Escape :=  "\" "u"  HexDigit HexDigit HexDigit HexDigit
```

Lo stream (sequenza) di caratteri Unicode viene trasformato mediante un'analisi lessicale in una sequenza di elementi lessicali: gli spazi bianchi, i commenti ed i Java Tokens; gli spazi bianchi ed i commenti servono a separare i tokens, gli elementi lessicali significativi. Usando la notazione BNF possiamo descrivere la grammatica che definisce la struttura lessicale nel seguente modo:

```
Input := [InputElements]
InputElements := {InputElement}
InputElement  := WhiteSpace | Comment |Token
WhiteSpace := ASCII LF character | ASCII CR character |CRLF | ASCII SP
character | ASCII HT character | ASCII FF character
Comment := /*text / | //a single-line comment | /** documentation comment */
Token := Identifier | Keyword | Literal |Separator |Operator
```

Non intendiamo qui definire tutti metasimboli sopra elencati (il metasimbolo `Comment` è definito per brevità in modo non formale) ma ci limitiamo ad osservare che per `Keyword` si intende una delle parole chiave del linguaggio, per `Separator` un elemento di punteggiatura, per `Operator` uno dei simboli di operazione, per `Identifier` un nome definito dall'utente, per `Literal` il nome proprio di elementi fondamentali (caratteri, numeri, valori booleani, stringhe, `null`). I letterali sono nomi propri costruiti mediante una regola semplice che consente di descrivere infiniti oggetti (es letterali per numeri interi, decimali).

La grammatica che descrive la struttura dei programmi Java sintatticamente corretti utilizza i Java token come elementi terminali.

Qualche Definizione

Formalizziamo alcune delle nozioni date e forniamo una definizione insiemistica di linguaggio.

Sia T = {a1,..., at} un insieme finito non vuoto di simboli detto **alfabeto**; una **parola** o **stringa** su T è una sequenza finita di elementi di T. Ad esempio, se T ={0,1}, allora 001, 100111110 sono parole su T. Il numero di simboli della sequenza è la lunghezza della stringa; la **stringa vuota**, che denotiamo con "", è la stringa di lunghezza 0, cioè la parola che non consiste in alcun simbolo. Due stringhe sono uguali se i loro caratteri, letti ordinatamente da sinistra a destra, coincidono.
Sia T+ l'insieme di tutte le sequenze finite di elementi di T. Definiamo una operazione binaria, denotata da #, detta concatenazione: se x e y sono stringhe appartenenti a T+, la concatenazione di y a x è la sequenza ottenuta giustapponendo x a sinistra e y a destra. Ad esempio se x=a1a2a3 e y=b2a3a1 x#y=a1a2a3b2a3a1. Per brevità scriveremo xy anziché x#y. E' evidente che la concatenazione è una operazione associativa (cioè (xy)z=x(yz)), quindi [T+, #] è un semigruppo, di solito chiamato semigruppo libero generato dall'insieme T+. L'aggettivo libero è usato per indicare che [T+, #] è privo di identità tra gli elementi di T+ tranne quelle che esprimono la proprietà associativa. Con T* si denota l'insieme di tutte le parole,

inclusa la parola vuota, su T. L'insieme T* con l'operazione di concatenazioen è un monoide dotato dell'identità bilatera "" (cioè ""x= x"" = x).

Un **linguaggio formale** L, di alfabeto T, è un insieme di parole su T; naturalmente L è contenuto propriamente o coincidente con T*. Esempi: T={a,b}, L1={ab, abb, bab}, L2={ax | x è una qualsiasi stringa di caratteri di T}, L1 è un linguaggio che contiene 3 parole, L2 è un linguaggio infinito le cui stringhe cominciano tutte con a.

Una **grammatica non contestuale Gr** è definita dai seguenti quattro elementi:

1. un insieme finito e non vuoto T di simboli terminali, detto **alfabeto terminale**;
2. un insieme finito e non vuoto N di simboli non terminali, o metasimboli, detto **alfabeto non terminale**; gli alfabeti N e T sono disgiunti e si definisce **alfabeto totale** *V* la loro unione.
3. un insieme P di **regole sintattiche** o **produzioni**; P è un insieme finito e non vuoto di coppie (A,B) dove la parte sinistra A appartiene a N (A è un simbolo non terminale) e la parte destra B appartine a *V**
4. un simbolo iniziale S, appartenente all'insieme dei simboli non terminali;

Una **forma di frase di Gr** (nel linguaggio generato dalla grammatica) è definita ricorsivamente nel modo seguente:

S è una forma di frase.

se ABC è una forma di frase e (B,D) è una produzione, anche ADC è una forma di frase. Si dice che ADC **deriva direttamente secondo G** da ABC. Se A1 deriva direttamente da A0 secondo G, A2 deriva direttamente da A1 secondo G, A3 deriva direttamente da A2 secondo G,... , Ak deriva direttamente da A(k-1) secondo Gr, si dice che Ak **deriva in k passi** da A0. In generale una stringa B deriva da una stringa A se esiste k>=0 tale che B deriva da A in k passi. Se k=0 allora A=B; k è **detto lunghezza della derivazione.**

Il **linguaggio non contestuale L(Gr)** generato dalla grammatica Gr, è costituito da tutte le sequenze di simboli terminali ottenibili dal simbolo iniziale S ed applicando via via le regole di produzione dell'insieme P. Altrimenti detto, L(Gr) è l'insieme di tutte le forme di frase di Gr che non contengono elementi di N cioè: L(Gr) = {x | x appartiene a T* AND x deriva da S secondo Gr}.

Alberi sintattici

La sequenza delle produzioni utilizzate per generare una stringa x da una grammatica G con simbolo iniziale S definisce la **struttura** di x, che potrebbe pertanto essere rappresentata da una delle derivazioni che da S consentono di generare x. Al fine di disporre di una rappresentazione univoca di x si utilizzano gli **alberi sintattici** , alberi ordinati, di seguito definiti. Consideriamo per esempio la grammatica `G=({S,H},{"a"},P,S)` dove `{S,H}è l'alfabeto non terminale, {"a"} è l'alfabeto terminale,, S è il simbolo iniziale` e `P denota le produzioni`:

```
S := H"a"
H := HS |"a"
```

la derivazione `S := H"a" := HS"a" := "a"S"a" := "a"H"aa" := "aaaa"` è rappresentata dall'albero sintattico S(H(H(a)S(H(a)a))a). Il processo di costruzione dell'albero sintattico associato ad una stringa prende il nome di **analisi sintattica** o **parsing**.

Un **albero ordinato** può essere definito ricorsivamente nel seguente modo: un albero A è un insieme finito, non vuoto, di vertici (nodi) tali che:

esiste un vertice speciale detto **radice** dell'albero;

esiste una partizione A1, A2,...,Am, con m >=0, degli altri vertici, tale che esista un ordinamento A1, A2,...,Am, dei blocchi e ogni sottoinsieme di vertici Ai, 1<=i<=m, sia a sua volta un albero.

Questa definizione ricorsiva non è circolare in quanto ogni sottoalbero Ai contiene meno vertici dell'albero A.
I blocchi Ai sono detti **sottoalberi** della radice. Inoltre si assume che con riferimento alla notazione grafica, se i<j, Ai è situato completamente a sinistra di Aj. I vertici privi di discendenti sono le **foglie**, gli altri sono detti **nodi interni.**

Un ***V*-albero** è un albero i cui vertici sono etichettati con elementi dell'insieme *V*. Nell'esempio mostrato sopra, la radice dell'albero sintattico è etichettata con S ed ha 2 figli o sottoalberi. Il primo figlio (quello più a sinistra) è a sua volta un albero di radice H, quello di destra consiste della sola radice ("a").
Viene detta **frontiera dell'albero** la stringa di simboli che etichettano le foglie, letti da sinistra a destra.

Per ogni grammatica non contestuale G e per ogni stringa x appartenente a L(G) esiste un albero sintattico di radice S, avente per frontiera la stringa x. Non siamo qui interessati agli algoritmi che data una stringa x e una grammatica non contestuale G costruiscono l'albero sintattico di x.

Esistono grammatiche che generano una stringa come frontiera di due alberi sintattici distinti. In tal caso si parla di **grammatiche ambigue**. Esempio: la grammatica definita dalle produzioni

```
E := E "+" E
E := "a"
```

è ambigua perchè la frase "a+a+a" ha 2 differenti alberi sintattici.

Nel linguaggio naturale l'ambiguità sintattica (esistono, infatti, anche altri tipi di ambiguità detti semantici) è un fenomeno intrinseco che si verifica assai di frequente. Per esempio "un dolce rosso" può essere una torta di colore rosso oppure un uomo rosso dal carattere dolce, questo se "dolce" e "rosso" sono definiti dalla grammatica sia come aggettivi che come sostantivi.
Agli effetti delle applicazioni ai linguaggi di programmazione l'ambiguità è una proprietà negativa. Infatti il significato di una frase può essere definito come una funzione dell'albero sintattico e, se ne esiste più di uno per una stessa frase, essa può avere un significato non univoco. Si preferisce pertanto cercare una grammatica differente che, pur definendo lo stesso linguaggio, non sia ambigua.

Un compilatore può essere visto come un algoritmo che, per ogni programma scritto in un linguaggio ad alto livello (Pascal, C, Java,…) , detto *linguaggio sorgente*, genera il programma corrispondente in un linguaggio,
detto *linguaggio oggetto*, che possa essere eseguito dal calcolatore.
Da un punto di vista logico il funzionamento del compilatore può essere decomposto in due fasi: il programma viene prima riconosciuto come appartenente al linguaggio sorgente quindi se ne esegue la traduzione nel corrispondente linguaggio oggetto. Il fatto che i costrutti (le frasi) dei linguaggi di programmazione siano definite con precisione da grammatiche non contestuali consente di fare un'analisi sintattica rigorosa e di appoggiare la generazione della traduzione alla descrizione strutturale delle frasi.

Sintassi astratta

La sintassi astratta identifica i componenti significativi di ogni costrutto del linguaggio, trascurando i dettagli di notazione. Consideriamo il seguente esempio:
a+b; (+ a b); ADD a TO b
Le tre rappresentazioni sono diverse ma la sintassi astratta, indipendente dalla notazione, è la stessa; i componenti significativi di ogni costrutto sono l'operatore di somma e gli operandi denotati con a e b. Le espressioni +ab, a+b, ab+, in forma prefissa, postfissa ed infissa hanno gli stessi componenti significativi. Come viene rappresentata la sintassi astratta? Mediante alberi o in forma BNF. Consideriamo un'espressione E formata da un operatore, OP, e dalle sottoespressioni E1, E2,..., Ek, per k>=0. Possiamo rappresentare E mediante un albero alla cui radice è associato OP, la radice ha k figli ad ognuno dei quali è associata una espressione Ei. Ad espressioni costituite da una costante o da una variabile corrisponderà una foglia.

Tipi di dati astratti

Lo sviluppo di programmi grandi e complessi, che richiedono una grande quantità di risorse umane per essere realizzati, ha reso evidente una cosa ovvia, e cioè che la complessità può essere affrontata solo mediante la modularizzazione. Modularizzare significa risolvere un problema complesso suddividendolo in sottoproblemi più semplici fino a giungere a problemi così semplici da poter essere risolti facilmente in modo corretto. E' pertanto necessario poter specificare come la soluzione di un sottoproblema si rela con le soluzioni degli altri. Lo studio delle tecniche di programmazione sembra indicare che il metodo più adeguato per affrontare il problema della modularizzazione consiste nell'esaminare la natura dei dati del problema che si intende risolvere. Questo metodo è alla base della programmazione a oggetti e si fonda sul concetto di tipo di dato astratto; vedremo che definire un oggetto equivale a definire un tipo di dato astratto.
Informalmente un tipo di dato astratto è costituito da uno o più insiemi e da operazioni su questi insiemi:
TIPO DI DATO ASTRATTO = INSIEMI + OPERAZIONI
Più precisamente un tipo di dato astratto è descritto da un nome, da una collezione di insiemi (domini) su cui sono definite operazioni ammissibili e da assiomi che specificano le proprietà delle operazioni.
Consideriamo come primo esempio due tipi di dati astratti ben noti; sono tipi fondamentali e sono pertanto predefiniti per la gran parte dei linguaggi di programmazione. Essi sono il tipo booleano ed i naturali.
Poichè la specifica deve essere data in modo indipendente dalla sintassi concreta di un particolare linguaggio di programmazione definiremo una semplice sintassi.

Il tipo booleano

Il tipo booleano, che denotiamo con `Bool`, è costituito da un insieme di due elementi (costanti), `true` e `false`, e dalle operazioni `NOT, AND, OR, XOR`. Nel seguito le costanti sono denotate come funzioni prive di dominio, e le variabili (`a,b,c, x,y, z, s, P`) sono da intendere quantificate universalmente.

```
Name Bool
Operations
      true: ()→ Bool;
      false: ()→ Bool;
      NOT: (Bool) → Bool;
      AND: (Bool, Bool) → Bool;
      OR: (Bool, Bool) → Bool;
      XOR:(Bool, Bool) → Bool;
Axioms x,y: Bool;
      NOT(true) = false; NOT(false) = true;
      AND(true,x)=x; AND(false,x)=false; AND(x,y)=AND(y,x);
      OR(true,x)= true; OR(false,x)= x; OR(x,y)= OR(y,x);
      XOR(x,y)= XOR(y,x);  XOR(y,x)= NOT(x=y);
```

Il tipo Nat

Consideriamo ora la specifica dei naturali con le operazioni di somma e prodotto e la relazione di ordinamento. Denotiamo i naturali con `Nat`; poiché per definirli abbiamo bisogno anche del tipo booleano diciamo che `Nat` usa il tipo `Bool`.

```
Name Nat
Uses Bool
Operations
      0: ()→ Nat;
      succ: (Nat) → Nat;
      isZero: (Nat) → Bool;
      _ + _ :(Nat,Nat) → Nat;
      _ * _ :(Nat,Nat) → Nat;
      _ <= _:(Nat,Nat) → Bool;
      _ < _ :(Nat,Nat) → Bool;
```

```
Axioms x,y,z: Nat; P: Nat → Bool;
       isZero(0)= true;
       isZero(succ(x))= false;
       succ(x)= succ(y)⇒ x=y;
       ((P(0)AND(P(x)⇒P(succ(x))) ⇒ P(y);
       (x+0)=x;
       (x+succ(y))= succ(x+y);
       (x*0)=0;
       (x*succ(y))= (x*y)+x);
       (x<=x) = true;
       (x<=y) AND (y<=x) ⇒ x=y;
       (x<=y) AND (y<=z) ⇒ (x<=z);
       (x<y)⇔(x<=y) AND NOT(x=y));
```

I primi tre assiomi sono gli assiomi di Peano per i numeri naturali; i quattro assiomi seguenti specificano le proprietà di somma e prodotto. Gli ultimi assiomi definiscono la funzione minore o uguale (<=) e la funzione minore (<). Il primo assioma asserisce che zero non è successore di alcun numero; il secondo asserisce che per ogni numero naturale c'è al più un predecessore, il successore del quale è il numero naturale. Il terzo infine formula il principio di induzione: per ogni predicato P, se vale P(0) e se P(x) implica P(succ(x)) allora P vale per ogni x.
Le operazioni `0 e succ` consentono di costruire tutti gli interi, le chiamiamo pertanto costruttori. Gli assiomi relativi alle operazioni di somma e prodotto sono dati in funzione dei costruttori.
Le proprietà associativa e commutativa di somma e prodotto e la proprietà del prodotto "esiste 1 tale che per ogni x, x*1=x" si derivano dagli assiomi sopra definiti.

Il tipo SetNat

Definiamo ora il tipo di dato astratto `SetNat` che specifica ogni sottoinsieme finito di naturali.

```
Name SetNat
Uses Nat, Bool
```

Operations

```
       empty: ()→ SetNat;
       insert:(SetNat,Nat) → SetNat;
       remove:(SetNat,Nat) → SetNat;
       isEmpty: (SetNat) → Bool;
       contains:(SetNat,Nat) → Bool;
Axioms s: SetNat; x,y: Nat
       isEmpty(empty) = true;
       isEmpty(insert(s,x)) = false;
       contains(empty,x)) = false;
       contains(insert(s,x),x) = true;
       contains(remove(s,x),x) = false;
       remove(insert(s,x),x) = s;
       remove(empty,x) = empty;
       insert(insert(s,x),x) = insert(s,x);
       insert(insert(s,x),y) = insert(insert(s,y),x);
```

Le operazioni `empty` and `insert` sono i costruttori; i primi cinque assiomi descrivono il comportamento delle operazioni `isEmpty` e `contains` rispetto ai costruttori. Gli ultimi due assiomi specificano che l'inserzione multipla equivale all'inserzione singola e che il risultato è indipendente dall'ordine di inserzione degli elementi. La specifica data può essere arricchita con la definizione delle operazioni di unione, intersezione, complemento e differenza tra insiemi.

Il difetto della specifica precedente è che essa è troppo particolare: riguarda solo insiemi di numeri naturali. E' necessario produrre una diversa specifica per ogni tipo di elementi di un insieme. Un'utile alternativa per produrre altre specifiche è fornire una **schema di specifica** come quello che segue, in esso con **`Item`** si intende un arbitrario tipo di dato astratto. Rispetto alla specifica precedente aggiungiamo un'altra funzione `size` che fornisce la cardinalità di un insieme.

```
Name SetItem
      Uses Bool, Nat, Item
Operations
            empty: ()→ Set;
            insert:(SetItem,Item) → SetItem;
            remove:(SetItem,Item) → SetItem;
            isEmpty: (SetItem) → Bool;
            contains:(SetItem,Item) → Bool;
            size: SetItem → Nat;
      Axioms s: SetItem; x,y: Item
            isEmpty(empty) = true;
            isEmpty(insert(s,x)) = false;
            contains(empty(),x)) = false;
            contains(insert(s,x),x) = true;
            contains(remove(s,x),x) = false;
            remove(insert(s,x),x) = s;
            remove(empty(),x) = empty;
            insert(insert(s,x),x) = insert(s,x);
            insert(insert(s,x),y) = insert(insert(s,y),x);
            size(empty)=0;
            size(insert(s,x))= if (contains(s,x)) then size(s), else size(s)+1;
            size(remove(s,x))= if (contains(s,x)) then size(s)-1, else size(s);
```

La funzione `if_then_ else:(Bool,Value,Value)→ Value` ha tre argomenti: il primo è di tipo booleano poiché `if` è seguito da un predicato, gli altri due argomenti hanno il tipo del valore delle espressioni che seguono `then` ed `else` rispettivamente. Il risultato dipende dalla valutazione del predicato.

Il tipo Stack

Consideriamo ora il tipo di dato astratto Stack (Pila), un tipo di dato fondamentale per eseguire computazioni.
Informalmente uno stack (pila) è una collezione finita di elementi disposti l'uno sull'altro come in una pila di piatti; gli elementi posti nello stack sono dello stesso tipo (es. uno stack di naturali, di interi, di stringhe...). Gli elementi possono essere aggiunti e rimossi dalla sequenza ad una sola delle due estremità, dalla cima, detta top. E' possibile conoscere solo l'elemento che è in cima alla pila.
La descrizione data fornisce un'idea di cosa è una pila; seppur non precisa, essa contiene tutti gli elementi che la specifica formale dovrà definire. Abbiamo parlato di "collezione di elementi", "disposti in sequenza" e di tre operazioni che interessano la pila: "aggiungere un elemento", "rimuovere un elemento", "leggere l'elemento in cima alla pila". Queste tre operazioni hanno i nomi di `push`, `pop`, `top` ed interessano solo la cima della pila:

`top` è una operazione il cui risultato è l'elemento in cima allo stack;
`push è una operazione che, dato un elemento ed uno stack, produce come risultato uno stack in cui l'elemento dato è inserito in cima;`
`pop` è una operazione che, dato uno stack, produce come risultato uno stack in cui l'elemento prima situato in cima è rimosso;

E' necessario introdurre altre due operazioni, una per creare uno stack vuoto, ed un'altra per verificare se uno stack è vuoto; chiameremo queste due operazioni `emptyStack` e `isEmptyStack`.
Anche in questo caso forniamo una specifica parametrica, denotando con `StackItem` uno stack di elementi di tipo **`Item`**.

```
Name StackItem
Uses Bool, Item
```

Operations

```
      emptyStack: ()→ StackItem;
      push: (Item, StackItem) → StackItem;
      top: (StackItem) → Item;
      pop: (StackItem)→ StackItem;
```

```
isEmptyStack: (StackItem)→ Bool;
```

La specifica data non dice nulla circa il significato delle operazioni, significato che deve essere definito mediante assiomi; essa tuttavia assume che tutte le operazioni producano come risultato un elemento di uno dei tre insiemi `StackItem`, `Item`, `Bool`. Riflettendo sull'operazione `top` ci si accorge però del fatto che questa assunzione non è corretta, infatti quando l' operazione `top` è applicata allo stack vuoto essa non può fornire come risultato un elemento di tipo `Item`.
Che cosa può fornire allora? La soluzione a questo problema è stata data aggiungendo ad ogni dominio su cui è costruito un tipo di dato astratto un particolare elemento il **null**; l'applicazione della operazione `top` ad uno `StackItem` vuoto fornirà pertanto il valore `null` del tipo `Item`. Quando si programma è sempre necessario verificare se l'operazione richiesta produce un risultato utile oppure un valore `null`, tale situazione deve essere infatti trattata in modo opportuno.
Qual'è il risultato dell'operazione `pop` su uno `StackItem` vuoto? Potremmo assumere che sia uno `StackItem` vuoto oppure il valore `null`; scegliamo la seconda alternativa.
Gli assiomi devono precisare le seguenti specifiche informali delle operazioni:

se si crea un nuovo `StackItem` mediante `emptyStack` e ad esso si applica l'operazione `isEmptyStack` si ottiene il valore `true`.
se ad uno `StackItem` si aggiunge un elemento di tipo `Item` mediante l'operazione `push` si ottiene uno `StackItem` che contiene almeno un elemento di tipo `Item`.
se ad uno `StackItem` si aggiunge un elemento di tipo `Item` mediante l'operazione `push` e ad esso si applica l'operazione `top` si ottiene l'elemento posto nello `StackItem`.
se ad uno `StackItem` si aggiunge un elemento di tipo `Item` mediante l'operazione `push` e ad esso si applica l'operazione `pop` si ottiene lo `StackItem` originale.
se ad uno `StackItem` vuoto si applica l'operazione `top` si ottiene come risultato il valore `null` di `Item`.
se ad uno `StackItem` vuoto si applica l'operazione `pop` si ottiene come risultato il valore `null` di `StackItem`.

```
Axioms: s: StackItem,x: Item
      isEmptyStack(emptyStack) = true;
      isEmptyStack(push(x,s)) = false;
      top(push(x,s)) = x;
      pop(push(x,s)) = s;
      top(emptyStack) = null;
      pop(emptyStack) = null;
```

Gli assiomi dati costituiscono una definizione precisa della semantica delle operazioni? Rispondere a questa domanda significa chiedersi se la definizione data è completa. Per i nostri scopi consideriamo informalmente il problema della completezza e consideriamo completo un insieme di assiomi che:

definisca i risultati di tutte le composizioni ammissibili di operazioni;
definisca le operazioni che consentono la costruzione di tutti i possibili esemplari del tipo di dato astratto.

Abbiamo definito uguale a `null` il risultato delle operazioni `top` e `pop` su uno stack vuoto; queste operazioni possono essere composte con altre, dobbiamo pertanto essere certi che il risultato della composizione sia sempre definito. A tal fine è necessario imporre che ogni operazione applicata ad un argomento di valore `null` fornisca come risultato `null`. Pertanto:

`top(pop(emptyStack))` fornisce come risultato `null`.

Infatti `pop(emptyStack())`produce come risultato `null` e l'operazione `top` con argomento `null` fornisce il valore `null`. La sequenza di operazioni:

`push(a,pop(push(b,push(c,emptyStack))))` fornisce come risultato uno stack contenente la sequenza `ac`.

Relativamente alla richiesta fatta di poter costruire tutte le possibili istanze di stack a partire dalle operazioni sopra definite osserviamo che:

le operazioni `emptyStack` e `push` consentono di costruire uno stack;
l'unica altra operazione che modifica uno stack è `pop;`
`top` e `isEmptyStack` non modificano il contenuto di uno stack.

Si può intuitivamente concludere che ogni stack può essere costruito da una composizione di `emptyStack`, `push` e `pop` e che tale composizione può sempre essere espressa come composizione delle sole operazioni `emptyStack` e `push`.
Espressioni relative agli stack che contengono solo le operazioni `emptyStack` e `push` sono dette “reduced expressions”.

Si può dimostrare che esiste una sola reduced expression che definisce un dato stack, tali espressioni sono pertanto dette essere in forma canonica.

Il tipo List

Un altro tipo di dato astratto fondamentale per le computazioni è la lista. Anche in questo caso, come per lo stack, introduciamo il modello di lista in modo informale ed intuitivo. Una lista è una sequenza finita di elementi di un certo tipo, un elemento può essere ripetuto nella lista e la lista può essere vuota. Una lista può essere ricorsivamente definita nel seguente modo:

la lista vuota è una lista;
la coppia (elemento, lista) è una lista;

```
Name ListItem
Uses Nat, Bool,Item
```

Operations

```
      emptyList: ()→ListItem;
      addToList: (Item, ListItem) → ListItem;
      head: (ListItem)→ Item;
      tail: (ListItem) → ListItem;
      length: (ListItem) → Nat;
      isEmptyList: (ListItem) → Bool;
      concatenate: (ListItem, ListItem) → ListItem;
Axioms: a: ListItem, x: Item
      isEmptyList(emptyList) = true;
      isEmptyList(addToList(x,emptyList)= false;
      head(emptyList)= null;
      tail(emptyList)= emptyList;
      head(addToList(x,a)= x;
      tail(addToList(x,a)= a;
      length(emptyList)= 0;
      length(addToList(x,a))=length(a)+1;
      concatenate(emptyList,a)= concatenate(a, emptyList)= a;
      concatenate(a,b)= addToList(head(a),concatenate(tail(a),b));
      concatenate(concatenate(a,b),c)= concatenate(a,concatenate(b,c));
```

Gli assiomi definiscono il risultato delle operazioni `head` e `tail` effettuate sulla lista vuota e su una lista arbitraria; l'operazioni `head` su una lista vuota produce come risultato il valore `null`.
La lunghezza della lista è definita ricorsivamente: la lunghezza della lista vuota è 0 e la lunghezza della lista ottenuta aggiungendo un elemento ad una lista "a" 1+ lunghezza di "a".
E' evidenziata la proprietà associativa della concatenazione ed il fatto che la lista vuota costituisce l'identità bilatera rispetto alla concatenazione.

Il tipo di dato astratto lista può essere arricchito di ulteriori operazioni. Si può definire l'accesso posizionale agli elementi, quindi ottenere un elemento in una data posizione, ottenere la posizione di un dato elemento, sostituire l'elemento in una data posizione.

Tipi in Java

Java è un linguaggio **fortemente tipato**, ciò significa che ogni variabile deve essere dichiarata di un certo tipo. Dichiarare una variabile di un certo tipo significa definire l'insieme di valori che essa può assumere e le operazioni che su tali valori si possono effettuare. La tipizzazione forte aiuta ad individuare errori in fase di compilazione.
In Java i tipi sono divisi in due categorie: i **tipi primitivi** e i **tipi reference**.
I tipi primitivi sono predefiniti ed i loro nomi sono tra le parole riservate, **keyword**, del linguaggio; ai tipi non primitivi è invece necessario dare un nome. La forma BNF che segue presenta la categorizzazione dei tipi in Java:

```
Type:= PrimitiveType | ReferenceType
PrimitiveType := NemericType | "boolean"
NumericType := IntegralType | FloatingPointType
IntegralType : = “char” | “short” | “byte” | “int” | “long”
FloatingPointType := “float” | “double”
ReferenceType := ArrayType | ClassType | InterfaceType
```

I metasimboli `ArrayType, ClassType e InterfaceType` verranno definiti in seguito.

Nel linguaggio ci sono nomi propri per i tipi (es: `boolean,int`, `long`,…), nomi propri per gli elementi di un tipo (es: `true`, `false`, 1, 2, 3,…) e nomi comuni, le variabili, per denotare elementi arbitrari di un tipo.
E' chi programma che introduce i nomi (nomi di tipi non primitivi, nomi di variabili, nomi di costanti) e lo fa mediante una **dichiarazione**; una dichiarazione introduce una nuova entità ed include sempre un identificatore che viene usato ogni volta che ci si deve riferire a quella particolare entità. Un identificatore è una sequenza non limitata di *Java letters* e *Java digits* che inizia sempre con una Java letter. Una Java letter è un carattere di un sottoinsieme dell'Unicode che comprende i caratteri maiuscoli e minuscoli dell'alfabeto latino, l'underscore _ e il dollaro $; un Java digit è un elemento di {0,1,…, 9}.Non si possono usare come identificatori le parole chiave del linguaggio, i letterali del tipo boolean, il letterale null. Java è case-sensitive.

La creazione di una variabile è fatta mediante la **dichiarazione di variabile** la cui sintassi è la seguente:

```
VariableDeclaration := TypeName VarIdentifier ";"
```

La dichiarazione di variabile arricchisce il linguaggio di un nome che non denota alcun valore. Per assegnare un valore ad una variabile è necessario effetture un **assegnamento** la cui sintassi è la seguente:

```
Assignment:= VarIdentifier “=” VarValue “;”
```

Le variabili non inizializzate non possono essere utilizzate in alcuna espressione. Se si usa una variabile senza aver assegnato ad essa un valore il compilatore segnala un errore.

E' possibile effettuare in un'unica frase la dichiarazione di più variabili dello stesso tipo e la dichiarazione di più variabili con assegnamento. Non c'è limite al numero di variabili che possono essere dichiarate in un'unica dichiarazione, se sono molte il programma risulta meno leggibile.

Consideriamo ora i tipi primitivi e mostriamo frasi ben formate del linguaggio.

Il tipo `boolean`

Il tipo `boolean` è il tipo di dato astratto che abbiamo definito. E' costituito da un insieme di due elementi, denotati dai letterali `true` e `false`, dalle operazioni AND, OR, OR Esclusivo, NOT, che in Java vengono denotate rispettivamente con i simboli &, |, ^, ! e dalle relazioni: == (uguale) != (diverso). Si possono pertanto scrivere frasi del tipo:

```
boolean z;
boolean x,y,z;
```

```
boolean x = true;
boolean y = false;
boolean x = true, y = false;
```

e dopo avere dichiarato che `x,y,z` sono di tipo `boolean` ed avere assegnato un valore ad `x` e ad `y` possiamo scrivere frasi quali:

```
z = x & y;
z = x | y;
z = x ^ y;
z = !(x | y);
boolean confronto = (z == x);
```

Tipi per rappresentare numeri interi

Ci sono quattro diversi tipi per rappresentare sottoinsiemi dei numeri interi. Essi differiscono per l'intervallo di valori che possono rappresentare:

`byte` rappresenta gli interi da -128 a 127 inclusi
`short` rappresenta gli interi da -32768 a 32767 inclusi
`int` rappresenta gli interi da -2147483648 a 2147483647 inclusi
`long` rappresenta gli interi da -9223372036854775808 a 9223372036854775807 inclusi

Nella Java Virtual Machine, gli interi sono rappresentati in notazione binaria nella forma detta complemento a due; ogni elemento di tipo `short`, `byte`, `int`, `long` richiede pertanto per essere memorizzato 8-bit, 16-bit, 32-bit e 64-bit ripettivamente.

Letterali

Gli interi dei quattro tipi possono essere denotati in notazione decimale, esadecimale (base 16) o ottale (base 8). Un letterale che denota un intero di tipo `long` deve avere come suffisso uno dei due caratteri L o l. In notazione BNF:

```
IntegerLiteral := DecimalIntegerLiteral | HexIntegerLiteral |
OctalIntegerLiteral
DecimalIntegerLiteral := DecimalNumeral [IntegerTypeSuffix]
DecimalNumeral := “0” | NonZeroDigit {Digit}
Digit := “0” | NonZeroDigit
NonZeroDigit := | “1” | “2” | “3” | “4” | “5” | “6” | “7” | “8” | “9”|
IntegerTypeSuffix := “L”  | “l”

HexIntegerLiteral := HexNumeral [IntegerTypeSuffix]
HexNumeral := “0x”HexDigit | “0X”HexDigit | HexNumeral HexDigit
HexDigit := “0” | “1” | “2” | “3” | “4” | “5” | “6” | “7” | “8” | “9”|
“a” | “b” | “c” | “d” | “e” | “f” | “A” | “B” | “C”| “D” | “E” | “F”
IntegerTypeSuffix := “L”  | “l”

OctalIntegerLiteral := OctalNumeral [IntegerTypeSuffix]
OctalNumeral := “0” OctalDigit | OctalNumeral OctalDigit
Octal Digit := := “0” | “1” | “2” | “3” | “4” | “5” | “6” | “7” | “8”
IntegerTypeSuffix := “L”  | “l”
```

Esempi di letterali `int`: `0` `2` `1999` `0372` `0xDadaCafe` `0x00FF00FF`
Esempi di letterali `long`: `0L` `68L` `2000l` `0372L` `0x10000000000L` `0xC0B0l`
I letterali che in notazione decimale denotano interi diversi da zero non possono iniziare con il digit “0”; e' evidente che 10.000 o 10,000 non sono letterali che denotano interi.

Dichiarazione di variabile e assegnamento

Secondo la sintassi definita le frasi che seguono sono corrette:

```
int x, y, x;
short aShort = 2000;
byte aByte, b = 100;
long aLong = 100L;
```

```
long x, y;
long aLong = 5000000000L;
```

Se una variabile di tipo `int` è inizializzata con un valore troppo grande, es:

```
int x = 5000000000;
```

il compilatore segnala l'errore "integer number too large".

Se vengono effettuati i seguenti assegnamenti:

```
byte aByte = 200;
short aShort = 50000;
```

il compilatore segnala l'errore "possible loss of precision: int required".

Conversioni tra tipi

E' sempre possibile convertire un valore di tipo `S` in un valore di tipo `T` se `S` è sottotipo di `T`; tali conversioni dette *widening conversions*, non comportano alcuna perdita di informazione. Consideriamo il codice:

```
short aShort = 20;
int x = aShort;
```

alla variabile `x` dichiarata di tipo `int` è assegnato un valore di tipo `short`.
Cosa accade se un valore di un `IntegralType T`, è assegnato ad un suo sottotipo `S`? In questo caso si parla di *narrowing conversions* e non si ha perdita di informazione solo se il valore del tipo convertito è anche un valore del sottotipo. Per effettuare una narrowing conversion è necessario effettuare una operazione di **casting** (parola che viene dalla metallurgia e che significa dare forma). L'operazione di casting ha la sintassi che segue:

```
CastExpression := "("IntegralType")"  Expression ";"
```

Il tipo dell'operando `Expression` viene convertito nel tipo esplicitamente nominato tra parentesi rotonde; il tipo di una `CastExpression` è pertanto il tipo il cui nome compare tra parentesi rotonde. Il codice:

```
int x = 20;
short aShort = x;
```

che assegna alla variabile `aShort,` dichiarata di tipo `Short,` un valore di tipo `int` è sbagliato. Il codice corretto richiede l'operazione di casting:

```
int x = 20;
short aShort =(short)x;
```

Il casting fa si che a run time un valore numerico di un certo tipo venga convertito in un valore numerico di un sottotipo e ciò può comportare errori. Nell'esempio che segue:

```
int x, y;
short z,k;
x = 10;
y = 50000;
z = (short)x;
k = (short)y;
```

assegnando a `z` il valore di `x` non si ha alcuna perdita di informazione, assegnando a `k` il valore di `y` si ha in `k` un valore errato (-15536).

Operazioni e relazioni binarie

Su tutti i tipi che rappresentano interi sono definite le operazioni aritmetiche:
`+` (somma) , `*`(prodotto), `-`(sottrazione), `/`(divisione intera), `%`(modulo)
e le relazioni: `==` (uguale) `!=` (diverso), `>` (maggiore), `<` (minore) `>=` (maggiore o uguale), `<=` (minore o uguale).

Le espressioni aritmetiche vengono di solito scritte in notazione infissa e possono essere scritte con parentesi rotonde o senza. Se sono presenti parentesi rotonde vengono valutate prima le espressioni nelle parentesi più interne poi, via via le altre. Se le espressioni sono scritte senza parentesi rotonde, ai fini della valutazione valgono le note regole di precedenza tra operazioni (`*`,`/`,`%` hanno la stessa precedenza e sono eseguite prima di `+`, `-`). I simboli delle operazioni binarie sono sintatticamente *left-associative* (si raggruppano da sinistra a destra: `a*b/c` equivale a `(a*b)/c`, le operazioni sono pertanto eseguite nell'ordine in cui compaiono nell'espressione.
La valutazione delle operazioni è fatta sempre prima della valutazione dei predicati `<, >, >=, <=,` che a loro volta sono valutati prima dei predicati `==` e `!=`. Tutti i simboli di relazione sono sintatticamente *left-associative.*

Ogni espressione denota un valore, ha pertanto un tipo, il tipo del valore denotato. Se il valore di un'espressione è di tipo `T` esso deve essere assegnato ad una variabile di tipo `T`. Di che tipo sono le espressioni della forma `(x op y)`, dove `op` è una variabile che denota una delle operazioni `+,*, -, /, %,` e `x` e `y` sono variabili di uno dei 4 tipi `byte`, `short`, `int`, `long`?

Le operazioni aritmetiche effettuate su valori di tipo `byte`, `short`, `int`, foniscono sempre un valore di tipo `int`; se almeno uno degli operandi è di tipo `long`, il risultato delle operazioni è di tipo `long`. Java converte automaticamante gli operandi di una operazione aritmetica ad un tipo comune, `long` se almeno uno degli operandi è di tipo `long, int` in tutti gli altri casi. Queste widening conversions vengono effettuate automaticamente perché i calcoli vengono eseguiti utilizzando 32 o 64 bit.

Osserviamo che è naturale chiamare somma l'operazione che somma due interi qualunque sia il loro valore, pertanto le operazioni:

```
_+_ (int,int) → int
_+_ (long,long) → long
```

diverse per il tipo degli argomenti e per il tipo del risultato, hanno lo stesso nome; ciò vale naturalmente anche per le altre operazioni. Quando viene dato lo stesso nome ad operazioni che operano su domini (insiemi di valori) diversi, si dice che si ha **overloading** dell'operazione.

Mostriamo ora frammenti di codice che utilizzano il tipo `boolean` e i tipi per rappresentare interi

```
int x, y, z;
x = 3 + 2;
y = 4*54+28-42;
z = (2*(44-1))/3;
boolean confronto = (3*2+5)>= 24;
int n;
x = n*7;
```

L'ultima frase è errata poiché alla variabile `n` non è stato assegnato alcun valore (`n` non è stata inizializzata), il compilatore segnala l'errore "variable n might not have been initialized".

Consideriamo la seguente espressione:

```
int result = 4*2000000000/4;
```

vorremmo ottenere come risultato `2000000000` ed invece otteniamo un risultato errato, l'intero negativo `-926258176 ((2^32 –8000000000) /4 = -926258176)`. Ciò perché la variabile `result` è dichiarata di tipo `int` ma 4*2000000000 non denota un intero di tipo `int` ma un intero di tipo `long`. E' stato commesso un **errore di overflow**; errori di questo tipo non vengono segnalati. In questo caso ci sono due soluzioni alternative: dichiarare `result` di tipo `long` o scrivere l'espressione usando le parentesi rotonde `4*(2000000000/4)`.

Se per errore accade che un intero sia diviso per un'espressione che valutata produce il valore 0, si blocca l'esecuzione del programma e viene segnalato l'errore "Java Lang Arithmetic Exception: / by zero". Ciò vale anche per l'operazione modulo. Il codice:

```
byte y=10 z=2;
byte x = y/z;
short s = 200;
short aShort = 10*s;
```

è sbagliato, infatti il risultato delle operazioni aritmetiche su valori di tipo `byte e short` è un valore di tipo `int` ed esso non può essere assegnato ad una variabile di tipo `byte` oppure `short.` In questi casi il compilatore segnala

l'errore: "possible loss of precision: int required". Tuttavia in questo caso è ragionevole fare una conversione da valori di tipo `int` a valori di uno dei sottotipi poiché la conversione non determina alcuna perdita di informazione. Il seguente codice è pertanto corretto:

```
byte y=10 z=2;
byte x = (byte) y/z;
short s = 200;
short aShort =(short)10*s;
```

Operazioni unarie

Su tutti i quattro tipi sono definite le operazioni unarie + e – ; il simbolo di operazione è prefisso. La semantica di `+operando` è il valore dell'operando, la semantica di `-operando` è il valore dell'operando è sottratto al valore `0.` I simboli delle operazioni unarie sono *right associative* (`a*b/-c = (a*b) /(-c)`).
Poiché l'ambito dei valori positivi e negativi non è simmetrico, applicare l'operatore – al massimo intero negativo di tipo `int` o `long` produce come risultato lo stesso numero negativo, si ha un errore di overflow che non viene segnalato (-(-2147483648) = -2147483648). Nel caso di `byte` o `short` si ha la promozione al tipo `int`, pertanto il codice:

```
byte x, y;
x = -128;
y = -x;
```

non è corretto; il compilatore sull'ultima istruzione segnala l'errore: "possible loss of precision: int required".

Il tipo char

Il tipo `char` è utilizzato per descrivere i caratteri; le sue costanti , i caratteri dell'Unicode, sono denotate da letterali, CharacterLiteral. Un CharacterLiteral è costituito da un carattere singolo posto tra apici o da una escape sequence posta tra apici. Più precisamente:

```
CharacterLiteral :=  ' SingleCharacter'  |  'EscapeSequence'
```

`Un` SingleCharacter è qualsiasi carattere Unicode ad eccezione di: ' (Single quote) , \ (Unicode Escape) e dei caratteri ASCII LF or CR che sono utilizzati per indicare la terminazione di una riga. Esempi di letterali che denotano valori di tipo carattere:

```
'a', '%', 't', '\u03a9', 'uFFFF'
```

le escape sequences:

`'\\'` (per rappresentare il backslash),
`'\''` (per rappresentare l'apice),
`'\"'` (per rappresentare il doppio apice che individua un letterale di tipo stringa).:
'\b': per rappresentare il backspace (ASCII BS character)
'\t' : per rappresentare l' horizontal tab (ASCII HT character)
'\n': per rappresentare il new line (ASCII LF character)
'\f': per rappresentare il form feed (ASCII FF character)
'\r': per rappresentare il carriage return (ASCII CR character)

Le ultime quattro escape sequences rappresentano gli spazi bianchi.

Poiché i caratteri Unicode sono posti in corrispondenza con i naturali (da 0 a 65355), su valori di tipo `char` sono definite le operazioni e le relazioni descritte per i tipi che rappresentano interi. Il risultato di tali operazioni, può essere assegnato ad una variabile di tipo `char` solo dopo una operazione di cast. Il codice che segue è pertanto errato:

```
char x,y,z;
x='a';
y='b';
z= x+1;
```

E' invece corretto scrivere:

```
char x,y,z;
x='a';
```

```
y='b';
z= (char) (x+1);
```

I tipi `float` e `double`

Per trattare numeri interi molto grandi e molto piccoli (non rappresentabili mediante il tipo `long`)e per trattare approssimazioni di numeri razionali e di numeri reali sono definiti i tipi primitivi `float` e `double`. Entrambi rappresentano un numero mediante una coppia di numeri: la mantissa e l'esponente; la mantissa definisce la precisione, l'esponente la scala. Pertanto, la rappresentazione di un numero non è posizionale ma in virgola mobile (floating point).

La definizione dell'insieme di valori rappresentabili mediante i tipi `float` e `double,` e la descrizione delle operazioni in virgola mobile che su di essi si possono effettuare è conforme allo *IEEE Standard for Binary Floating Point Arithmetic*, ANSI/IEEE Standard 754-1985. Gli intervalli di valori sono:

da `-3.40282347*10^38` a `+3.40282347*10^38` per il tipo `float`
da `-1.79769313486231570*10^308` a `+1.7*79769313486231570^308` per il tipo `double`

Il più piccolo valore positivo diverso da 0 di tipo `float` è 1.40239846*10^-45
Il più piccolo valore positivo diverso da 0 di tipo `double` è 4. 9406564584124654*10^-324

Letterali

Un valore `float` è rappresentato mediante 4 byte, la mantissa è rappresentata con un'accuratezza di 7 digit, un valore `double` è rappresentato mediante 8 byte, la mantissa è rappresentata con un'accuratezza di 17 digit.

Per denotare elementi di entrambi i tipi si utilizzano FloatingPointLiterals. Un letterale che denota valori floating point è costituito da una parte intera, un punto, una parte frazionaria, un esponente ed un suffisso. Nella parte intera o nella parte frazionaria deve essere presente almeno un digit e sono richiesti almeno il punto o l'esponente o il suffisso. Un FloatingPointLiteral di tipo `float` deve contenere il suffisso `f` o `F`, per il tipo `double` il suffisso è opzionale.
Esempi di letterali `float`:

```
1e1f     2.f     .3f     0F     3.14f    6.002345e+23f
```

Esempi di letterali `double`:

```
1e1      2.      .3      0.0    3.14    1e-9d    1e137
```

I letterali per valori floating point prevedono solo la rappresentazione in base 10.
Forma BNF:

```
FloatingPointLiteral :=  {Digit} “.”[{Digit}] [ExponentPart]
[FloatTypeSuffix]
| “.” {Digit}  [ExponentPart] [FloatTypeSuffix]
| {Digit} ExponentPart [FloatTypeSuffix]
| {Digit} [ExponentPart] FloatTypeSuffix

ExponentPart := ExponentIndicator SignedInteger
SignedInteger := [Sign] {Digit}
Sign := “+” | “-”
ExponentIndicator := “e” | “E”
Digit := “0” | “1” | “2” | “3” | “4” | “5” | “6” | “7” | “8” | “9”|
FloatTypeSuffix := “f”  | “F” | “d”  | “D”
```

Dichiarazione di variabile e assegnamento

Secondo la sintassi definita le frasi che seguono sono corrette:

```
float x, y, x;
double z, d = 20000.0;
x = 6.002345e+23f;
z = d;
```

Se, come nel codice che segue, una variabile di tipo `float` è inizializzata con un valore troppo grande o con un letterale di tipo double, il compilatore segnala l'errore "possible loss of precision: double required":

```
float f = 20000.0;
float f = 1e137;
```

Operazioni e relazioni

Su entrambi i tipi sono definite le operazioni binarie:
`+` (somma) , `*`(prodotto), `-`(sottrazione), `/`(divisione), `%`(floating point remainder);
le relazioni binarie:
`==` (uguale) `!=` (diverso), `>` (maggiore), `<` (minore) `>=` (maggiore o uguale), `<=` (minore o uguale);
le operazioni unarie: `+, -`

La semantica dell'operazione floting point remainder è la seguente: `op1 % op2` fornisce il resto dopo aver diviso `op1` per `op 2` un numero intero di volte; es: 3 % 1.2 produce come risultato 0.6.
Nella valutazione delle espressioni valgono le usuali regole di precedenza (vedi quanto detto sopra per i tipi interi).
Occorre porre attenzione all'uso del predicato `==`, spesso è sensato chiedersi se due valori differiscono al più di una quantità fissata e non se sono lo stesso valore (non ha senso chiedersi se due rappresentazioni approssimate di un numero reale sono uguali).

E' necessario osservare che oltre ai numeri positivi e negativi lo standard prevede: lo zero positivo (0.0), lo zero negativo (-0.0), e + infinito (`Infinity`), - infinito (`-Infinity`) per i valori rispettivamente maggiori del massimo e minori del minimo valore degli intervalli di definizione; inoltre è definito uno speciale valore, *Not-a-Number* (NaN), per rappresentare le forme di indeterminazione (es: 0.0/0.0). I valori `Infinity` , `–Infinity e NaN` non possono essere assegnati ad alcuna variabile.

Ad eccezione del valore NaN i valori floating point sono ordinati (-infinito, valori finiti negativi, -0.0, 0.0, valori finiti positivi, + infinito); se comparati, lo zero positivo e lo zero negativo risultano uguali (`0.0 == -0.0` è valutato `true`, `0.0 >= -0.0` è valutato `false`) ma altre operazioni possono distinguere tra i due valori (es: 1.0/0.0 produce come risultato `Infinity`, 1.0/-0.0 produce come risultato `-Infinity`).
Poiché NaN non è ordinato le comparazioni `>, >=, <, <=,` producono come risultato `false` se entrambi o uno degli argomenti è NaN; la comparazione `==` produce come risultato `false` se uno dei due argomenti è NaN e `!=` produce come risultato `true` se uno dei due argomenti è NaN.

Eseguendo calcoli con numeri floating point può accadere di trovarsi in due condizioni di errore:
1) il calcolo produce come risultato un valore che non può essere rappresentato perché maggiore del massimo o minore del minomo valore rappresentabile, in tal caso il risultato è `Infinity` o `–Infinity.` La somma, la sottrazione o il prodotto di un valore finito con un valore non finito (`Infinity` o `–Infinity`) produce come risultato `Infinity` o `–Infinity.`Qualsiasi valore diviso per `Infinity` o `–Infinity` produce come risultato 0.
2) se nel calcolo si giunge ad una forma di indeterminazione cioè al valore NaN, il valore si propagherà e contaminerà il risultato di tutte le espressioni in cui esso è utilizzato.

Si possono scrivere espressioni che contengono variabili di qualsivoglia tipo numerico, il risultato della valutazione delle espressioni deve però essere assegnato ad una variabile di un tipo corretto. Consideriamo il codice che segue:

```
double x;
int k = 1;
short s = 10;
byte r = 3;
x= k*r/s;
```

Alla variabile `x` di tipo `double` può essere assegnato il valore di tipo `int` che risulta dalla valutazione dell'espressione; occorre però osservare che poiché le variabili `r` e `s` sono promosse al tipo `int` (attraverso una widening conversion automatica) e la variabile `k` è di tipo `int`, la divisione eseguita è la divisione intera. Se chi scrive il codice intende realizzare la divisione deve fare il cast al tipo `double` di una delle variabili, per esempio:

```
x= (double)k*r/s;
```

oppure

```
x= (double)(k)*r/s;
```

In entrambi casi k è forzata al tipo double, l'operazione di cast ha una priorità maggiore di quella delle operazioni aritmetiche.
Qui abbiamo fatto il cast da un tipo ad un supertipo, ciò è sempre possibile ma è necessario tener presente che se la conversione da float a double non determina alcuna perdita di informazione (come quelle tra gli Integral Type) le conversioni da int o long a float e da long a double possono in qualche caso determinare la perdita di qualcuno dei bit meno significativi.

E' ragionevole fare il cast di un tipo ad uno dei sottotipi solo nel caso in cui, come nell'esempio che segue, non si abbia una perdita di precisione:

```
int x =1000;
double z = 10.000;
float y = 10000f;
short result = (short)((x*z)/y);
```

L'esecuzione del codice:

```
double x,y,z;
z = 1e304;
y = 1e100;
x = y*z;
```

produce un valore non contenuto nell'intervallo ammissibile per il tipo double, il valore di x sarà Infinity.

Stenografia

Poiché le operazioni di incrementare e decrementare di 1 un valore numerico vengono eseguite di frequente, al fine di rendere il codice più efficiente è consentito denotare tali operazioni mediante i simboli (++), (--) che possono essere sia prefissi che postfissi all'espressione da incrementare o decrementare di 1. Se usati prefissi tali simboli sono sintatticamente right-associative, se usati postfissi sono left-associative. Diamo la sintassi e la semantica di queste operazioni medianti esempi.
Consideriamo il caso prefisso; il codice:

```
int x,y;
x = 7;
y = ++x;
```

ha la stessa semantica del codice:

```
int x,y;
x = 7;
x = x+1;
y = x;
```

In maniera analoga, il codice:

```
int x,y;
x = 7;
y = --x;
```

ha la stessa semantica del codice:

```
int x,y;
x = 7;
x = x-1;
y = x;
```

Quando l'operatore è prefisso, il valore della variabile viene incrementato di 1 (o decrementato di 1) ed il nuovo valore viene utilizzato nell'operazione che deve essere eseguita, negli esempi sopra illustrati è l'operazione di assegnamento.
Il codice:

```
int x,y;
x = 7;
y = 2*++x;
```

ha la stessa semantica del codice:

```
int x,y;
x = 7;
x = x+1;
y = 2*x;
```

Quando l'operatore è postfisso il valore della variabile viene incrementato di 1 (o decrementato di 1) solo dopo aver utilizzato la variabile. Pertanto:

```
int x,y;
x = 7;
y = x++;
```

ha la stessa semantica del codice:

```
int x,y;
x = 7;
y = x;
x = x+1;
```

e:

```
int x,y;
x = 7;
y = 2*x++;
```

ha la stessa semantica del codice:

```
int x,y;
x = 7;
y = 2*x;
x = x+1;
```

Osserviamo che le operazioni unarie di incremento e decremento di 1 applicate a valori di tipo `byte, short` e `char` producono rispettivamente valori di tipo `byte`, `short` e `char.` E' pertanto corretto scrivere:

```
short x,y;
x = 7;
y = x++;
```

ed anche:

```
char x,y, z;
 x='a';
 y='b';
 z= x++;
```

Anche qui ci sono problemi di overflow che non vengono segnalati:

```
byte x,y;
x = 127;
y = x++;
```

il valore di `x` dopo l'assegnamento è `-128`.

Queste operazioni sono definite per ognuno dei tipi numerici sopra definiti (`char, byte, short, int, long, float, double`).

Operazioni bitwise

Tutti i valori di tipi `IntegralType` (`byte, short, char, int, long`) sono rappresentati nella macchina da parole binarie, su tali parole sono definite operazioni bit a bit, *operazioni bitwise,* che operano su tutti i bit della sequenza. Esse sono:

-) le operazioni binarie AND, OR e OR Esclusivo, denotate rispettivamente mediante i simboli &, |, ^

-) l'operazione unaria complemento, denotata dal simbolo ~

-) le operazioni di shift a sinistra, di shift a destra propagando il segno da sinistra (signed right shift), e di shift a destra sostituendo 0 al valore spostato (unsigned right shift), denotate rispettivamente dai simboli <<, >>, >>>.

Le operazioni binarie AND, OR, OR Esclusivo, combinano i bit corrispondenti dei due operandi, esse possono essere utili quando si desidera utilizzare un intero per rappresentare la presenza o l'assenza di certe condizioni, per esempio:

```
int condizioni = 6 ;
int condizione1 = 1;
condizioni = condizioni | condizione1;
```

il risultato di questo codice è di porre ad 1 il primo bit, a partire da destra, nella sequenza che rappresenta il valore assegnato alla variabile `condizioni`.

L'operazione unaria complemento trasforma il valore 0 in 1 e viceversa.

Le operazioni di shift richiedono di precisare di quante posizioni deve essere effettuato uno shift e se esso debba essere effettuato a sinistra o a destra. Consideriamo come esempio la rappresentazione binaria del numero 8 realizzata mediante 8 bit (00001000) e modifichiamo la stringa di bit spostando tutti i bit di una posizione a sinistra (00010000) e di una posizione a destra (00000100). Il risultato di queste operazioni è la rappresentazione binaria dei numeri 16 e 4, lo shift a sinistra equivale infatti a moltiplicare il numero per due, lo shift a destra a dividerlo per due (naturalmente si tratta della divisione intera). Le operazioni di moltiplicazione e divisione intera realizzate mediante uno shift sono molto efficienti.

Fare uno shift a sinistra richiede di inserire uno zero a destra, viceversa nel caso in cui si faccia uno shift a destra ci sono due possibilià: inserire uno zero a sinistra (unsigned right shift) o mantenere il bit che rappresenta il segno (signed right shift), tale differenza è rilevante per i numeri negativi. Se data la rappresentazione binaria del numero -8 realizzata mediante 4 bit (1000) si effettua uno shift di una posizione a destra mantenendo il segno si ottiene –4 (1100), a partire da –6 (1010)si ottiene –3 (1101).

Il numero di posizioni di cui effettuare lo shift deve essere un numero positivo (con un numero negativo il compilatore non segnala errore ma si ottiene come risultato 0), tale numero deve essere compreso tra 0 e 31 nel caso in si operi su parole di 32 bit (valori di tipo `int, byte, short, char)` e tra 0 e 63 nel caso in si operi su parole di 64 bit (valori di tipo `long).` Se si usano valori più grandi il compilatore non segnala errore ma in tal caso si ottiene uno shift corrispondente al numero modulo 32 o modulo 64. La sintassi dell'operazione di shift richiede che a destra del simbolo dell'operazione si ponga il numero di posizioni di cui deve essere effettuato lo shift. Es:

```
int x, y;
x = 256;
y = x >> 8;
y = y << 31;
x = y >>>24;
```

Il codice:

```
int x, y;
x = 1;
y = x << 31;
```

produce lo stesso risultato del codice:

```
int x, y;
x = 1;
y = x << 63;
```

cioè il massimo intero di tipo `int`.

Strutture di controllo

Il codice Java è costituito da **statement**; uno statement è una frase del linguaggio (sequenza ben formata di elementi del linguaggio) che denota un'azione.
Ci sono diversi tipi di statement; il più semplice è lo statement vuoto, a cui non corrisponde nessuna azione. La sua sintassi è:

```
EmptyStatement := ";"
```

Ogni statement termina con un punto e virgola e, in assenza di indicazioni differenti, gli statement vengono eseguiti nell'ordine in cui sono scritti. Possiamo distinguere statement di base e strutture più articolate che chiamiamo blocchi. Un blocco (Block) è una sequenza di statement racchiusa tra parentesi graffe. La scelta di usare parentesi graffe per delimitare un blocco aiuta la compilazione; infatti se si usassero parentesi rotonde o quadrate l'analisi sintattica sarebbe più difficile poiché le prime sono usate nelle espressioni, le seconde servono a denotare gli array.

```
Statement := EmptyStatement | BaseStatement | BlockStatement
BlockStatement := "{" {Statement} "}"
```

Un `BlockStatement` è uno statement che contiene altri statement, per `BaseStatement` si deve intendere qualsiasi costrutto elementare del linguaggio. Sino ad ora abbiamo incontrato i costrutti di dichiarazione di variabile, di assegnamento e le regole per scrivere espressioni sui tipi primitivi, descriviamo ora gli statement di base che consentono di modificare l'ordine sequenziale di esecuzione delle azioni, essi sono denominati **strutture di controllo**. Le strutture di controllo sono utilizzate per effettuare la **selezione** e l' **iterazione condizionata** di statement. Le parole chiave di Java utilizzate nelle strutture di controllo sono:

`if, if-else, switch-case,` per la selezione
`for, while, do-while,` per l'iterazione condizionata

Strutture per la selezione

if-then statement

La parola chiave `if` è usata in uno statement la cui sintassi è:

```
IfThenStatement := "if" "("condition")" Statement
```

dove con `condition` si intende un'espressione che valutata produce un valore booleano.
La semantica del costrutto è la seguente: se la valutazione della condizione produce il valore `true` viene eseguito lo statement che segue la condizione altrimenti si passa allo statement che successivo. Es:

```
if (number % 2 != 0) number = number + 1;
```

Nel caso in cui lo statement che segue la condizione sia un blocco, ci sono due convenzioni per disporre le parentesi graffe in modo da rendere il codice il più leggibile possibile; in ogni caso è opportuno indentare gli statement del blocco rispetto all'`if`:

```
if (condition) {
  statement1;
  statement2;
  statement3;
}
```

oppure

```
if (condition)
{
  statement1;
  statement2;
  statement3;
}
```

Le parentesi graffe sono opzionali per un singolo statement, il codice risulta più leggibile se si mettono sempre.
Cosa accade se non si racchiude il blocco tra parentesi graffe?

```
if (condition) statement1;
  statement2;
  statement3;
```

`Statement1` viene eseguito solo nel caso in cui la valutazione della condizione produca come risultato `true,` gli statement successivi vengono eseguiti sempre.
Se si deve modellare una situazione che esprime `N` scelte indipendenti si avrà codice del tipo:

```
if (condition1) statement1;
if (condition2) statement2;
if (condition3) statement3;
…
if (conditionN) statementN;
```

if-then-else statement

Accade spesso di essere interessati ad esprimere una scelta tra due azioni alternative, in tal caso è necessario usare il costrutto `if-else`; tale costrutto consente di effettuare una scelta tra due statement eseguendo l'uno o l'altro ma non entrambi.

```
IfThenElseStatement:= "if" "("condition")" Statement "else" Statement
```

La semantica del costrutto è la seguente: se la valutazione della condizione produce come risultato `true,` viene eseguito lo statement che segue la condizione altrimenti viene eseguito lo statement che segue la parola chiave `else`. Dopo l'esecuzione della condizione espressa da una delle due alternative viene eseguito lo statement successivo. Es:

```
if (number % 2 == 0) x+1;
else {
   statement1;
   statement2;
 }
```

Cosa accade se si scrive il codice che segue?

```
if (condition1)
if (condition2) {…do something…}
else {…do somethingDifferent…}
```

Il blocco `{… do somethingDifferent…}`è eseguito quando risulta falsa `condition1` o quando risulta falsa `condition2`? In Java, questo problema, denominato del *dangling else* (else ciondolante), è risolto imponendo che il blocco venga eseguito se è falsa `condition2`. In altre parole l'`else` è legato all'`if` più interno. E' tuttavia buona norma evitare di scrivere codice poco leggibile, l'uso di parentesi graffe consente di precisare in modo non ambiguo le azioni da eseguire. Es:

```
if (condition1) {
  if (condition2) {…do something…}
  else {…do somethingDifferent…}
}
```

oppure:

```
if (condition1) {
  if (condition2) {…do something…}
}
else {do…somethingDifferent…}
```

Lo statement che segue un `else` è un qualunque statement; quindi può a sua volta essere un `IfThenStatement` o un `IfThenElseStatement`.

Analisi di casi
Se ci si trova a dover modellare la scelta tra `n` azioni alternative (`n` casi diversi) si può usare una sequenza di costrutti `else if.` Es:

```
int testScore;
char grade;
if (testScore >= 90)  grade = 'A';
else if (testScore >= 80) grade = 'B';
else if (testScore >= 70) grade = 'C';
else if (testScore >= 60) grade = 'D';
else grade 'E';
next statement;
```

Switch statement

Se la scelta tra le diverse alternative dipende dal valore di una variabile di tipo `char, byte, short`, `int` , è possibile usare un altro costrutto del linguaggio. Il costrutto utilizza le parole chiave `switch` e `case` ed ha la sintassi che segue:

```
SwitchStatement := "switch" "("Expression")" SwitchBlock
SwitchBlock := "{" {SwitchLabel Statement} "}"
SwitchLabel := "case" ConstantExpression " : " | "default : "
```

Utilizziamo un esempio per dare la semantica del costrutto. Es:

```
byte mese = (byte) (2*3-1);
switch (mese) {
   case 1: {stampa: il mese di gennaio ha 31 giorni;}
   case 2: {stampa: il mese di febbraio ha 28 giorni;}
   case 3: {stampa: il mese di marzo ha 31 giorni;}
   case 4: {stampa: il mese di aprile ha 30 giorni;}
   …
   case 12: {stampa: il mese di dicembre ha 31 giorni;}
}
```

La semantica è la seguente: in funzione del valore della variabile che segue la parola chiave `switch`, seleziona il caso (`case`) contrassegnato dalla costante che corrisponde a tale valore ed esegui il blocco di statement ad esso associato. Nell'esempio, alla variabile `mese` viene assegnato il valore 5 viene quindi viene selezionato il caso contrassegnato dalla costante 5 e di conseguenza viene stampata la frase "il mese di maggio ha 31 giorni". Cosa accade dopo l'esecuzione dello statement associato all'etichetta `case 5`? Vengono analizzati tutti i casi successivi.
Affinché ciò non accada, ed in generale si vuole questo quando deve scegliere una tra più alternative, è necessario far seguire ogni statement corrispondente ad una `SwitchLabel` dallo statement `break`. L'effetto di `break` è quello di trasferire il controllo al più interno costrutto `switch` che lo contiene e che viene in tal modo completato. Riscriviamo pertanto il codice dell'esempio sopra:

```
byte mese= (byte)(2*3-1);
switch (mese) {
   case 1: {stampa: il mese di gennaio ha 31 giorni; break;}
   case 2: {stampa: il mese di febbraio ha 28 giorni; break;}
   case 3: {stampa: il mese di marzo ha 31 giorni; break;}
   case 4: {stampa: il mese di aprile ha 30 giorni; break;}
   …
   case 12: {stampa: il mese di dicembre ha 31 giorni; break;}
}
```

L'ultimo `break` non è necessario e si può omettere. Nel caso in cui per ogni caso ci sia un solo statement seguito da `break` le parentesi graffe sono opzionali.

Poiché i mesi sono dodici e ad una variabile di tipo byte possono essere assegnati i valori [-128,…,127] è opportuno prevedere cosa fare nel caso in cui alla variabile mese sia assegnato un valore non utilizzato in nessuna dei dodici casi di interesse. In tale situazione infatti nessuno statement dello `SwitchBlock` può essere eseguito, verrà eseguito lo statement che segue lo `SwitchStatement.` Per l'utente tuttavia è importante sapere cosa è accaduto, è quindi necessario trattare anche le situazioni non corrispondenti ai casi di interesse. A tal fine Java utilizza la parola chiave `default` che, posta dopo l'ultimo `case`, consente di definire un'azione da effettuare in tutte le condizioni non previste dai casi. Nell'esempio sopra uno statement da aggiungere potrebbe essere:

```
default : {stampa: il numero non è un valido indicatore di mese;}
```

Non avremmo potuto usare il costrutto `switch` per scrivere codice quale quello sopra descritto per determinare i voti, poiché il grado viene assegnato se il voto è maggiore od uguale ad un certo valore e non se il valore è uguale.

Per riassumere ricordiamo che la sintassi di uno `SwitchStatement` deve verificare le seguenti condizioni affinchè il compilatore non segnali un errore:

la costante che segue la parola chiave `case` deve essere del tipo della variabile che segue la parola chiave `switch;` inoltre solo i tipi `char, byte, short`, `int` sono ammessi
la stessa costante non può essere usata per segnalare casi diversi;
uno `SwitchStatement` deve contenere una sola etichetta `default.`

Conditional operator

E' un operatore ternario che verifica il valore di un'espressione booleana per decidere quale di due espressioni deve essere valutata. Ha la sintassi descritta dalla seguente forma BNF:

```
ConditionalExpression:= BooleanExpression "?" Expression ":" Expression";"
```

Se la condizione espressa dall'espressione booleana è vera, il risultato della ConditionalExpression è il risultato della valutazione del secondo operando cioè l' espressione che segue il punto interrogativo; se la condizione espressa dall'espressione booleana è falsa il risultato è dato dalla valutazione del terzo operando, l' espressione che segue i due punti.
Ci sono vincoli sul tipo delle espressioni che costituiscono il secondo ed il terzo operando, esse infatti devono essere entrambe di tipo booleano, oppure entrambe di tipo numerico, oppure entrambe di tipi reference compatibili o di tipo nullo. Il codice:

```
int x,y,z;
x=10;
y=24;
z= x<y ? y-x : x-y;
```

equivale al codice:

```
int x,y,z;
x=10;
y=24;
if(x<y) y-x;
else z=x-y;
```

L'operatore "?" ":" è sintatticamente right-associative (l'espressione `a?b:c?d:e?f:g` equivale a `a?b:(c?d:(e?f:g))`.

Strutture per l'iterazione

Il linguaggio Java dispone di tre costrutti che consentono di iterare azioni, essi utilizzano le parole chiave `while`, `do-while` e `for`.

while statement

Il costrutto che utilizza la parola chiave `while` ha la sintassi che segue:

```
WhileStatement := "while" "(" condition ")" Statement
```

dove `condition` è un'espressione di tipo booleano.
La semantica del costrutto è la seguente:

se la valutazione di `condition` produce come risultato `true,` esegui `Statement` e rivaluta `condition`
se la valutazione di `condition` produce come risultato `false,` esegui lo statement successivo.

Pertanto, se la condizione risulta falsa alla prima valutazione, `Statement` non viene mai eseguito; in caso contrario viene eseguito sino a quando la condizione permane vera.
Esempi di codice:

```
int i=1, sum =0, limit =20;
while (i<= limit) {
  sum = sum+i;
  i = i+1;
}
```

Osserviamo che se il valore di `i` non viene incrementato il programma non termina:

```
int i=1, sum =0, limit =20;
while (i<= limit) {
  sum = sum+i;
}
```

do-while statement

Il costrutto che utilizza le parola chiave `do` e `while` ha la sintassi che segue:

```
DoWhileStatement := "do" Statement "while" "(" condition ")" ";"
```

dove `condition` è un'espressione di tipo booleano. La semantica del costrutto è la seguente:

esegui `Statement` e valuta la condizione espressa da `condition`
se la valutazione di `condition` produce come risultato `true,` esegui di nuovo `Statement` e rivaluta `condition`
se la valutazione di `condition` produce come risultato `false,` esegui lo statement successivo.

In questo costrutto lo `Statement` che segue la parola chiave `"do"` è eseguito almeno una volta. Es:

```
int i=1, sum =0, limit =20;
do {
  sum = sum+i;
  i = i+1;
}
while (i<= limit);
```

L'esecuzione del codice porta ad avere `sum = 210`; eseguendo il codice utilizzato per esemplificare il costrutto `while` si ottiene lo stesso risultato. Se la variabile `i` viene inizializzata con il valore `21` il valore assegnato alla variabile `sum` sarà `0` nel primo caso e `21` nel secondo caso.

for statement

Il costrutto che utilizza la parola chiave `for` ha la sintassi che segue:

```
ForStatement := "for" "(" [ForInit] ";" condition ";" [ForUpdate] ")"
Statement
```

Per `ForInit` si intende una sequenza di espressioni separate da virgole o la dichiarazione di una o più variabili con assegnamento, con `ForUpdate` si intende una sequenza di espressioni separate da virgole. Le espressioni del `ForInit` e `ForUpdate` sono valutate in sequenza da sinistra a destra.
Il costrutto `for` ha la semantica seguente:
1) valuta la lista di espressioni che costituisce il `ForInit;` se il `ForInit` non è presente non viene eseguita alcuna azione;
2) valuta `condition` che deve essere di tipo `boolean:`

se la valutazione produce come risultato `true,` esegui `Statement`, valuta la lista di espressioni del `ForUpdate`, se presente, ed inizia un'altra esecuzione del ciclo `for` (valuta `condition`…).

se la valutazione produce come risultato `false,` esegui lo statement successivo al `ForStatement.`

Pertanto nel caso in cui `condition` venga valutata `false` la prima volta che si esegue l'iterazione descritta da un costrutto `for`, lo statement in esso contenuto non verrà mai eseguito.

Esempi di codice:

```
int i, sum =0, limit =20;
for (i = 1; i <= limit; i= i+1)
  sum = sum +i;
```

oppure dichiarando la variabile `i` all'interno del `ForInit`:

```
int sum =0,limit =20;
for (int i=1; i <= limit; i= i+1)
  sum = sum +i;
```

Abbiamo utilizzato il costrutto `for` per eseguire un'azione che sopra abbiamo eseguito mediante un costrutto `while`. Il costrutto `while` è più generale, conviene utilizzare il costrutto `for` quando ci si trova a dover eseguire uno statement un numero definito di volte perché esso consente di impostare il valore di un contatore e di incrementarlo per tener conto del numero di iterazioni effettuate. Es:

```
int i=0;
while(i<28) {
  do something;
  i=i+1;
}
```

Utilizzando il costrutto `for` possiamo scrivere:

```
for(int i=0; i<28; i=i+1){
  do something;
}
```

Il costrutto `for` consente di utilizzare una lista di espressioni nel `ForInit` e nel `ForUpdate`, es:

```
int sum =0,limit =20;
for (int i=1, int j=0; i <= limit; i= i+1, j= j+1)
  sum = sum + (i*j);
```

Esempio di codice per il calcolo del fattoriale:

```
int limit =10;
long factorial = 1L;
for (int j=2; j <= limit; j=j+1)
  factorial = factorial*j;
```

Spesso le espressioni del `ForUpdate` incrementano o decrementano di 1 un contatore; in tali casi per ragioni di efficienza si usano spesso le operazioni unarie `++` e `--`. Riscrivendo l'esempio precedente avremo:

```
int limit =10;
long factorial = 1;
for (int j=2; j <= limit; j++)
  factorial = factorial*j;
```

Codice per verificare se un numero è primo:

```
//alla variabile number, di tipo int, è assegnato un intero maggiore di 1

boolean prime = true;
for(int divisor = 2; divisor<(number-1); divisor++){
  if(number/divisor == 0) prime = false;
}
// stampare il risultato
```

Utilizzando un costrutto `while` invece di un costrutto `for` si può si può terminare l'iterazione appena viene individuato un divisore del numero:

```
//alla variabile number, di tipo int, è assegnato un intero maggiore di 1


boolean prime = true;
int divisor = 2;
while(prime & (divisor < number)){
  prime = number/divisor != 0;
  divisor = divisor+1;
}
// stampare il risultato
```

Statement che consentono di interrompere l'esecuzione di un blocco

Ci sono situazioni in cui può essere utile o necessario impedire l'esecuzione di una o più azioni previste in un blocco. Abbiamo già incontrato una situazione di questo tipo analizzando il costrutto `switch`; in esso, per uscire dal costrutto, abbiamo utilizzato uno statement la cui sintassi è:

```
BreakStatement := "break;"
```

Oltre che nel costrutto `switch`, il `BreakStatement` può essere contenuto in ognuno dei costrutti per l'iterazione (`while, do-while, for`), in tal caso interrompe la iterazione. Esempio:

```
…
int i;
for(i=0; i<n; i++){
   if(i*i=x) break;
}
…
```

L'effetto di un `BreakStatement` è sempre quello di trasferire il controllo all'esterno dello statement che lo contiene e questo può essere solo uno `SwitchStatement`, un `ForStatement`, uno `WhileStatement`, un `DoWhileStatement`, in caso contrario il compilatore segnala errore.

Nel caso in cui in uno dei costrutti per l'iterazione sia necessario uscire dal passo corrente per continuare l'iterazione si può utilizzare un `ContinueStatement` la cui sintassi è:

```
ContinueStatement := "continue;"
```

Esempio:

```
int sum =0,limit =20;
for (int = 1; i <= limit; i= i+1) {
  if(i%3 == 0)continue;
  sum = sum +i;
}
```

In questo caso al valore della variabile `sum` viene sommato il valore di `i` solo nel caso in cui questo non sia un multiplo di `3`.

In generale codice del tipo:

```
for(int i=0; i<n; i++) {
  if(something hold) {
    …do something…
  }
}
```

può essere scritto, usando un `ContinueStatement,` nel modo seguente:

```
for(int i=0; i<n; i++) {
  if(! something hold) continue;
```

```
            …do something…
        }
```
In questo modo si elimina un livello di indentazione.
Un `ContinueStatement` può essere contenuto solo in un blocco di statement di uno dei costrutti per l'iterazione (`while, do-while, for`), in caso contrario il compilatore segnala un errore.

Ci sono altri due costrutti del linguaggio, che utilizzano le parole chiave `break` e `continue` seguite da un identificatore, essi consentono di "saltare" dallo statement che si sta eseguendo ad un altro. Sono costrutti equivalenti al `go to` di altri linguaggi di programmazione, non li consideriamo perché essi sono da non utilizzare. Nei primi anni 70 è stato dimostrato che programmi che contengono istruzioni di salto possono essere riscritti utilizzando esclusivamente le strutture di controllo per la sequenza, la selezione e l'iterazione. Se non si utilizzano le strutture per il salto i programmi risultano modulari, più comprensibili e meglio verificabili. Per questa ragione la programmazione strutturata, programmazione che utilizza solo le strutture di sequenza, selezione ed iterazione, si è imposta come metodologia per la stesura dei programmi.

Ambito di utilizzo di una variabile

Il contesto di validità, **scope**, di una variabile è la parte di codice dove la variabile viene dichiarata ed utilizzata e determina quando la variabile viene eliminata. Consideriamo il codice:

```
        for(int i=0 …){
          …
        }
```
e
```
        int i;
        for(int i=0; …){
          …
        }
```
Nel primo esempio, il contesto della variabile `i` è quello racchiuso tra le parentesi graffe, solo nel codice racchiuso tra le parentesi graffe si può utilizzare `i`; nel secondo esempio il contesto di `i`inizia con la sua dichiarazione prima del `ForStatement` ed `i` può essere utilizzata anche al di fuori del contesto `for`.

Oggetti e Classi

Ad esclusione degli elementi dei tipi primitivi ogni elemento di cui parla il linguaggio Java è un oggetto. Vediamo pertanto di precisare cos'è un oggetto, come si definisce, come si comporta, come si usa, come si distrugge.

Se ci riferiamo al mondo reale è chiaro cosa intendiamo per oggetti; sono oggetti una casa, una penna, un libro, un albero, una persona, una città,…; inoltre è naturale raggruppare gli oggetti in classi: case, penne, libri, alberi, persone, città. Dicendo "una casa" abbiamo implicitamente dichiarato che esiste un tipo casa di cui una casa è un esemplare.

Gli oggetti software sono modelli degli oggetti del mondo reale e sono descritti definendone lo stato ed il comportamento. Lo stato di un oggetto software è descritto da **variabili**; le azioni eseguibili da un oggetto sono descritte da **metodi**.
Definire una classe consiste nel definire le variabili ed i metodi comuni a tutti gli oggetti della classe. In Java ogni classe è un **tipo** e gli oggetti della classe sono esemplari (istanze) del tipo, valori del tipo.

Utilizziamo un esempio semplice per introdurre i concetti che ci servono e definiamo una classe per rappresentare i punti nel piano cartesiano. Realizzeremo il seguente modello:

- un punto ha due coordinate: x,y;
- un punto è in grado di verificare se è uguale o diverso da un altro punto;
- un punto è in grado di effettuare una traslazione.

Dichiarazione di una classe

Per definire una nuova classe dobbiamo fare una **dichiarazione di classe**:

```
class Punto {… corpo della classe …}
```

`Punto` è il nome della classe; segue il corpo della classe, esso deve contenere:

- definizione dello **stato**
- definizione dei **costruttori**
- definizione del **comportamento** degli oggetti di tipo `Punto.`

Nonostante si possa scegliere qualsivoglia identificatore, e' convenzione attribuire alla classi identificatori che iniziano con una lettera maiuscola.

Stato

Lo stato di un punto consiste di una coppia di coordinate reali, dobbiamo pertanto rappresentarle mediante due variabili di tipo `double`:

```
class Punto {
  double x; // la coordinata x del punto
  double y; // la coordinata y del punto
}
```

E' convenzione attribuire alle variabili identificatori che iniziano con una lettera minuscola.

Costruttori

Elemento fondamentale di ogni classe sono i **costruttori**; i costruttori possono essere visti come particolari procedure che generano gli oggetti (esemplari) di una classe. I costruttori hanno lo stesso nome della classe e possono avere oppure no parametri formali. Un costruttore di punti dovrà inizializzare le variabili x e y.
Un costruttore per la classe `Punto`:

```
Punto (double aX, double aY){… corpo del costruttore …}
```

Esso ha come parametri formali due valori `aX` e `aY` di tipo `double` che verranno assegnati alle variabili `x` e `y` del nuovo punto creato dal costruttore.

Mettiamo nel corpo della classe il codice che definisce il costruttore:

```
class Punto {
  double x, y;    //variabili

  // costruttore
  Punto (double aX, double aY){
    x=aX;
    y=aY;
  }
}
```

Un costruttore viene invocato (la sua azione viene richiesta) mediante l'operatore `new`, parola chiave del linguaggio, seguita dal nome del costruttore, e da una coppia di parentesi rotonde. Se, come nel nostro esempio, il costruttore ha una lista di parametri formali, le parentesi rotonde devono contenere una lista di argomenti il cui numero e tipo deve corrispondere a quello della lista di parametri formali (nel nostro esempio due valori di tipo `double`). Es:

```
Punto p; //viene dichiarata una variabile di tipo Punto, non denota niente
p=new Punto(4.5,7.2);   //la variabile denota un punto con certe coordinate
```

I due statement precedenti si possono condensare in un unico statement:

```
Punto p = new Punto (4.5,7.2);
```

Comportamento (metodi)

Abbiamo descritto come un oggetto della classe `Punto` può essere costruito. Definiamo ora le due operazioni, **metodi**, che un oggetto della classe `Punto`, costruita per rappresentare il modello di punto sopra definito, deve eseguire.
Ogni metodo deve essere dichiarato; la **dichiarazione di un metodo** è costituita da un intestazione (header) e da un corpo (body). Nell'header viene definito il nome del metodo, il numero ed il tipo dei parametri formali, se ce ne sono, ed il tipo del risultato; quest'ultimo deve precedere il nome del metodo. La **segnatura** (signature) di un metodo consiste del nome, del numero e del tipo dei parametri formali.

Chiamiamo `equals` un metodo mediante il quale un punto confronta se stesso con un altro punto e lo considera "uguale" se ha le medesime coordinate:

```
boolean equals (Punto p) {
  if(p==null)return false;
  return (x==p.x & y==p.y);
}
```

La notazione `p.x` significa "la variabile `x` di `p` ". Se `p` è un termine di tipo `Punto, p.x` denota la variabile `x` di `p; p.y` denota la variabile `y` di `p; p.equals(p2),` dove `p2` è un termine di tipo `Punto`, denota un valore booleano.
Il metodo `equals` produce come risultato un valore booleano che viene reso disponibile a chi ha invocato il metodo mediante la paroala chiave `return` seguita da un'espressione del tipo corretto (un valore booleano).

Chiamiamo `translateOf` un metodo mediante il quale un punto modifica le sue coordinate effettuando una traslazione. Il metodo `translateOf` non rende alcun valore e dichiara ciò mediante la parola chiave `void`.

```
void translateOf (double deltaX, double deltaY) {
   x=x+deltaX;
   y=y+deltaY;
}
```

In un metodo che non produce in uscita alcun valore lo statement `return;` (`return` non seguito da alcuna espressione) è opzionale.
Mettendo insieme tutto ciò che abbiamo visto sino ad ora otteniamo una classe `Punto` molto semplice che rappresenta quanto specificato dal modello :

```
class Punto {
  double x, y;    //variabili

  // costruttore
  Punto (double aX, double aY){
    x=aX;
    y=aY;
  }

  //metodi
  boolean equals (Punto p) {
    if(p==null) return false;
    return (x==p.x & y==p.y);
  }
  void translateOf (double deltaX, double deltaY) {
    x=x+ deltaX;
    y=y+ deltaY;
  }
}
```

Il codice che definisce una classe deve essere memorizzato in un file il cui nome è costituito dal nome della classe seguito da `.java`, nel nostro caso il nome sarà `Punto.java`.

Il termine this

Immaginiamo di scrivere il metodo `translateOf` con parametri `x` e `y`. Ci troveremmo a dover distinguere `x` e `y`, variabili di stato di un oggetto di tipo `Punto`, da `x` e `y`, parametri formali del metodo.
Occorre una regola che consenta di distinguere di quale variabile si sta parlando, una regola che consenta di esprimere:

```
void translateOf (double x, double y){…
  … il mio x = il mio x + parametro x
  … il mio y = il mio y + parametro y
}
```

In Java l'ambiguità si risolve con l'uso del termine `this`:

```
void translateOf (double x, double y){…
  this.x = this.x +x;
  this.y = this.y +x;
}
```

`this.x` significa proprio "il mio `x` ".
Il tipo di `this` è il tipo della classe; nell'esempio `this` è di tipo `Punto`.

Un esempio

Ciò che intendiamo modellare è un segmento, un oggetto che ha le seguenti caratteristiche:

- un segmento ha due estremi che sono punti;
- un segmento è in grado di costruire il suo punto medio;
- un segmento è in grado di confrontarsi con un altro segmento;
- un segmento è in grado di traslarsi.

La classe `Segmento` utilizza come componenti oggetti di tipo `Punto.`

```
class Segmento {

  // variabili
  Punto estr1;   //un estremo
  Punto estr2;   //l'altro estremo

  // costruttori
```

```
    Segmento (Punto p1, Punto p2){
      estr1=p1;
      estr2=p2;
    }
    Segmento (Segmento s) {
      if(s!= null) {
        estr1=s.estr1;
        estr2=s.estr2;
      }
    }
    // metodi
    Punto puntoMedio(){
      if(estr1==null || estr2==null)return null;
      double middleX = (estr1.x + estr2.x)/2;
      double middleY = (estr1.y + estr2.y)/2;
      return new Punto(middleX, middleY);
    }
    boolean equals (Segmento s){
      if(s == null)return false;
      return(isPuntoEqual(estr1, s.estr1)&& isPuntoEqual(estr2, s.estr2));
    }
    boolean isPuntoEqual(Punto p1, Punto p2){
      if(p1==null) return (p2==null);
      return p1.equals(p2);
    }
    void translateOf (double deltaX, double deltaY){
      if(estr1 == null || estr2 == null)return;
      estr1.translateOf (deltaX, deltaY);
      estr2.translateOf (deltaX, deltaY);
    }
  }
```

La classe `Segmento` ha due costruttori, uno di essi ha come parametri formali due valori di tipo `Punto`, l'altro ha come parametro formale un valore di tipo `Segmento`. Entrambi i costruttori potrebbero essere invocati con un argomento di valore `null`, un punto inesistente o un segmento inesistente. E' necessario considerare espressamente questa situazione e trattarla in modo adeguato.
Se il primo costruttore viene invocato con due argomenti di valore `null`, esso assegnerà ai due estremi `estr1` e `estr2` il valore `null`, costruirà pertanto un segmento costituito da due punti di valore `null`; se uno solo dei due argomenti ha valore `null`, esso assegnerà all'estremo corrispondente il valore `null`, all'altro estremo un punto.
Se il secondo costruttore viene invocato con argomento `null`, non può fare nulla; al segmento inesistente `s` non ha senso chiedere il valore dei suoi estremi `s.p1` e `s.p2`; i termini `s.p1` e `s.p2` sono privi di significato. Il fatto che il segmento non sia stato creato dovrebbe essere reso noto, vedremo in seguito come fare.

Osservazione: una variabile di tipo `Segmento` a cui è assegnato il valore `null` non denota alcun segmento. Diverso è il segmento ai cui estremi è assegnato il valore `null`; esso esiste, i suoi estremi non esistono.

Entrambi i costruttori possono costruire un segmento i cui estremi sono punti nulli; occorre tenere conto di questo fatto nella scrittura dei metodi della classe.

Il metodo `puntoMedio` calcola il punto medio di un segmento operando sulle coordinate degli estremi, il metodo rende un oggetto di tipo `Punto`. Se uno o entrambi gli estremi sono nulli, restituisce un punto nullo.

Il metodo `equals` verifica se un segmento si ritiene uguale ad un altro segmento; esso confronta i propri estremi con gli estremi dell'altro segmento considerando i casi di estremi nulli. A tal fine usa un metodo ausiliario `isPuntoEqual`.

Il metodo `translateOf` trasla il segmento traslando entrambi gli estremi se non nulli.

Breve riepilogo

La dichiarazione di una **classe** definisce un nuovo tipo.
La dichiarazione dice cos'è un oggetto della classe e che cosa esso può fare (che metodi può eseguire). Nella dichiarazione della classe, e solo in essa, sono descritte le caratteristiche comuni a tutti gli esemplari della classe; esse sono il tipo ed il nome delle variabili di stato ed i metodi. Variabili e metodi sono denominati componenti, **members**, di una classe, le variabili sono denominate anche **fields**.
La dichiarazione di una classe contiene i **costruttori** degli esemplari della classe. I costruttori non sono members.
Gli esemplari vengono creati utilizzando la parola chiave **`new`** seguita dall'invocazione di un costruttore; la parola chiave `new` esprime il fatto che si costruisce qualcosa di nuovo, che prima non esisteva.
Solo ad oggetti che esistono si può chiedere l'esecuzione di un metodo.
Nel corpo dei costruttori e dei metodi si può utilizzare la variabile `this` per riferirsi all'oggetto di cui si sta parlando.

Attributi di accesso

Le classi `Punto` e `Segmento` ci sono servite per introdurre alcuni concetti fondamentali della programmazione a oggetti, tuttavia ci sono altri importanti argomenti da considerare.
Un programma object oriented è costituito da oggetti che interagiscono, un oggetto esegue un'operazione quando riceve una richiesta dall'esterno (uno dei suoi metodi viene invocato). Affinchè ad un oggetto possa essere fatta la richiesta di eseguire una certa operazione deve essere noto, **pubblico**, il fatto che l'oggetto possa farla; per contro è opportuno che tutto ciò che non riguarda il comportamento pubblico di un oggetto venga mantenuto nascosto, **privato**.
In altre parole è fondamentale individuare quali informazioni devono essere mantenute riservate, private e quali possono essere rese pubbliche; ciò è vero sia per i members (variabili e metodi) sia per i costruttori.
In generale si puo' asserire che:

- è opportuno che lo stato di un oggetto sia modificabile solo dall'oggetto stesso, ciò significa che le variabili di stato non devono essere accessibili dall'esterno, esse devono essere dichiarate private;
- il mondo esterno all'oggetto deve avere a disposizione tutta e sola l'informazione necessaria ad interagire con l'oggetto, i metodi che possono essere invocati dall'esterno devono essere dichiarati pubblici;
- l'oggetto può agire sul suo stato interno (sulle variabili private) direttamente o mediante metodi privati.

Le classi devono essere dichiarate pubbliche; in contesti molto particolari, che incontremo, può essere utile dichiarare una classe privata.
Riscriviamo pertanto la classe `Punto` dichiarando private le sue variabili e pubblici il suo costruttore ed i suoi metodi:

```
public class Punto {
  private double x, y;    //variabili private

  // costruttore pubblico
  public Punto (double aX, double aY){
    x=aX;
    y=aY;
  }

  //metodi pubblici
  public double getX(){
    return x;
  }
  public double getY(){
    return y;
  }
  public boolean equals (Punto p) {
    if(p==null)return false;
    return(x==p.x && y==p.y);
  }
  public void translateOf (double deltaX, double deltaY) {
    x=x+deltaX;
    y=y+deltaY;
  }
}
```

Le coordinate di un oggetto di tipo `Punto`, se dichiarate private, non sono direttamente accessibili dall'esterno, ciò significa che un oggetto che non appartiene alla classe `Punto` non potrà riferirsi alle coordinate di un punto `p` con termini del tipo `p.x e p.y`.
Se si ritiene utile o necessario consentire ad oggetti non appartenenti alla calsse di conoscere il valore di variabili private, quali sono le coordinate di un punto nel nostro esempio, si può risolvere il problema dichiarando opportuni metodi pubblici. I due metodi pubblici `getX()`e `getY()`,che abbiamo aggiunto alla classe `Punto,` forniscono rispettivamente il valore della variabile privata `x` e della variabile privata `y`. E' prassi consolidata utilizzare il verbo `get` per metodi che forniscono il valore di variabili private.
Se si vuole consentire la possibilità di assegnare un valore a variabili private è necessario dichiarare opportuni metodi pubblici; nel caso della classe `Punto` essi potrebbero essere:

```
public void setX(double aX){
  x=aX;
}
public void setY(double aY){
  y=aY;
}
```

oppure se più correttamente si ritiene che modificare un punto richieda di assegnare i valori di entrambe le coordinate si può dichiarare il metodo pubblico:

```
public void moveTo(double aX, double aY) {
  x=aX;
  y=aY;
}
```

Osserviamo che i metodi `setX, setY, moveTo, translateOf` rendono un punto modificabile

Abbiamo modificato la classe `Punto` definendo l'accessibilità dei suoi componenti e del suo costruttore; dobbiamo chiederci ora che influenza hanno questi cambiamenti sulle classi che utilizzano elementi della classe `Punto`, nel nostro piccolo mondo la classe `Segmento`.
Il metodo `puntoMedio` è l'unico metodo che si riferisce alle variabili private di elementi di tipo `Punto`, deve pertanto essere modificato; gli altri metodi, che invocano metodi pubblici della classe `Punto`, non richiedono nessun cambiamento. Riscriviamo la classe `Segmento` modificando il corpo del metodo `puntoMedio` ed aggiungendo gli attributi di accesso; dichiariamo pubblici tutti i metodi ad esclusione di `isPuntoEqual,` metodo ausiliario costruito per rendere il metodo `equals` semplice e robusto.

```
public class Segmento {

  // variabili private
  private Punto estr1, estr2 // i due estremi

  // costruttori pubblici
  public Segmento (Punto p1, Punto p2){
    estr1=p1;
    estr2=p2;
  }
  public Segmento (Segmento s) {
    if(s!= null) {
      estr1=s.estr1;
      estr2=s.estr2;
    }
  }

  //metodi pubblici
  public Punto puntoMedio(){
    if(estr1==null || estr2==null)return null;
    double middleX = (estr1.getX() + estr2.getX())/2;
    double middleY = (estr1.getY() + estr2.getY())/2;
    return new Punto(middleX, middleY);
  }
```

```
  public boolean equals (Segmento s){
    if(s == null)return false;
    return(isPuntoEqual(estr1, s.estr1)&&
           isPuntoEqual(estr2, s.estr2));
  }
  public void translateOf (double deltaX, double deltaY){
    if(estr1 == null || estr2 == null)return;
    estr1.translateOf (deltaX, deltaY);
    estr2.translateOf (deltaX, deltaY);
  }

  //metodo privato
  private boolean isPuntoEqual(Punto p1, Punto p2){
    if(p1==null) return (p2==null);
    return p1.equals(p2);
  }
}
```

Riepilogo sull'utilizzo degli attributi di accesso
E' buona norma di programmazione dichiarare private le variabili che descrivono lo stato degli oggetti; infatti, consentire l'accesso diretto ad esse ad opera di metodi di classi diverse potrebbe portare gli esemplari di una classe in uno stato inconsistente; nei nostri semplici esempi ciò non è evidente.
Al contrario, devono essere dichiarati pubblici i metodi ed i costruttori che si vogliono rendere disponibili ad elementi che non appartengono alla classe.
Se una componente (variabile o metodo) di una classe viene dichiarata privata, l'unico modo per accedere ad essa è attraverso un opportuno metodo pubblico dichiarato nella classe stessa.

Metodi e variabili `static`

Abbiamo affermato che una classe è un prototipo di cui gli oggetti sono esemplari. Questo è ciò che accade per la gran parte delle classi ma ci sono situazioni in cui questo modello non è adeguato a rappresentare la realtà che si vuole considerare. Ci sono situazioni in cui è necessario costruire classi di utilitità che forniscono servizi direttamente, non attraverso loro esemplari. Sono classi di cui non possono essere creati esemplari.
Un esempio significativo di questa esigenza è costituito dalla classe **`Math`**, costruita per fornire strumenti per il calcolo di funzioni matematiche di utilizzo frequente (radice quadrata, elevamento a potenza, funzioni trigonometriche, …).

La classe `Math` non contiene costruttori; non servono poiché non c'è alcun oggetto di tipo `Math` da costruire.
La classe consiste di un elenco di metodi pubblici eseguibili solo dalla classe stessa.

I metodi eseguibili da una classe sono detti **class methods,** per contro i metodi eseguibili dagli esemplari di una classe sono detti **instance methods** (tutti i metodi definiti per le classi `Punto` e `Segmento` sono instance methods).
I metodi eseguibili da una classe sono detti anche **static methods** perché definiti usando la parola chiave **`static`**.

Un metodo `XXX` della classe `Math` è invocato come `Math.XXX`, seguito dalla lista, eventualmente vuota, di argomenti. La dichiarazione del metodo per il calcolo della radice quadrata, definito nella classe `Math`, è la seguente:

```
public static double sqrt(double a)
```

Usiamo il metodo `Math.sqrt` per dichiarare due metodi da aggiungere alla classe `Punto` , essi calcolano la distanza euclidea tra due punti:

```
public class Punto {…
  public double distanceFrom(Punto p) {
    if(p==null) return –1; //la distanza è sempre >=0
    double dx = x-p.x;
    double dy = y-p.y;
    return Math.sqrt(dx*dx + dy*dy);
  }
  …
```

```
}
```

Questo metodo calcola la distanza di un punto, quello che esegue il metodo, da un altro punto, il punto rappresentato dal parametro formale `p`. Il metodo tratta il caso in cui `p` sia nullo ponendo il valore della distanza uguale ad un valore negativo (-1); ciò fa capire che si è in una situazione di errore. Situazioni di questo tipo possono essere più correttamente trattate con l'uso di strumenti che incontreremo (trattamento delle eccezioni).

Nel caso in cui si voglia programmare una funzione per il calcolo della distanza che abbia come argomento due oggetti di tipo `Punto`, nessuno dei quali esegue il metodo, occorre definire un metodo eseguibile dalla classe `Punto` e non dai suoi esemplari. Sarà pertanto un metodo di classe e dovrà essere dichiarato `static`:

```
public class Punto {…
  public static double distance(Punto p1, Punto p2){
    if(p1==null || p2==null) return –1; //la distanza è sempre >=0
    double dx = p1.x - p2.x;
    double dy = p1.y – p2.y;
    return (Math.sqrt(dx*dx + dy*dy));
  }
  …
}
```

Scriviamo un altro metodo eseguibile dalla classe `Punto` e non dalle sue istanze, un metodo `equals` che abbia come argomento due punti.

```
public class Punto {…
  public static boolean equals(Punto p1, Punto p2) {
    if(p1==null) return (p2==null);
    return p1.equals(p2);
  }
  …
}
```

Dopo aver verificato che il punto `p1` non è nullo, è lecito chiedere a `p1` di verificare se esso è uguale a `p2,` mediante il metodo `equals` precedentemente definito.

Avendo a disposizione questo metodo della classe `Punto` possiamo scrivere un metodo `equals` della classe `Segmento` che non richiede alcun metodo ausiliario privato; infatti utilizzando il metodo `equals` con due argomenti siamo in grado di considerare la situazione in cui gli estremi del segmento possono essere nulli:

```
public boolean equals (Segment s){
  if (s==null) return false;
  return Punto.equals(estr1, s.estr1) & Punto.equals(estr2, s.estr2);
}
```

Possiamo pertanto sostituire il metodo `equals` precedentemente definito per la classe `Segmento` ed eliminare il metodo privato `isPuntoEqual`. L'eliminazione di un metodo privato non ha alcuna conseguenza sul codice delle classi che utilizzano oggetti di tipo `Segmento` poiché esse hanno accesso solo a metodi pubblici.

Avendo a disposizione due metodi, uno di classe ed uno di istanza, per calcolare la distanza euclidea tra due punti possiamo estendere il nostro modello di segmento ed asserire che un segmento è in grado di calcolare la sua lunghezza. Dichiariamo pertanto un nuovo metodo che chiamiamo length:

```
public double length(){
  return Punto.distance(estr1, estr2);
}
```

In certe situazioni è necessario considerare la possibilità di definire variabili o costanti specifiche della classe e non delle sue istanze, si parla di **class variables.** Per contro le variabili che caratterizzano gli esemplari di una classe sono dette **instance variables**. Le variabili che abbiamo dichiarato nella classe `Punto` e nella classe `Segmento` sono instance variable, solo queste descrivono lo stato di un oggetto.

Perché si dovrebbero usare class variables? Supponiamo di aver bisogno di contare quanti esemplari di una certa classe sono stati creati, il modo più semplice di farlo è di definire una variabile della classe e di modificarne il valore ogni

volta che viene generata un'istanza della classe. Si ottiene una variabile di classe dichiarando la variabile `static`, per questa ragione le class variables sono dette anche **static variables.** Se una variabile è dichiarata `static` esiste una sola variabile indipendentemente dal numero di esemplari della classe che vengono costruiti ed essa è accessibile da tutti gli esemplari della classe.

Possono essere definite anche costanti specifiche di una classe. In Java una costante è considerata una variabile non modificabile, si esprime ciò utilizzando la parola chiave **`final`** nella dichiarazione della variabile, la dichiarazione prevede anche l'assegnamente del valore. La parola chiave `final` sta ad indicare che il valore assegnato variabile non può essere modificato, è finale.
Nella classe `Math` sono dichiarate due costanti di tipo `double` i cui nomi sono rispettivamente `E` e `PI`, esse sono usate per rappresentare un'approssimazione dei numeri reali "e" e Π. Esse sono dichiarate nel seguente modo:

```
public static final double E  = 2.718281828459045
public static final double PI = 3.141592653589793
```

La loro definizione nella class `Math` consente di utilizzare i termini `Math.E` e `Math.PI`, per riferirsi a queste costanti.
La classe `Math` è una classe pubblica che contiene solo metodi di classe pubblici e costanti (variabili finali) di classe pubbliche.

Il tipo "Array di"

Abbiamo considerato i tipi primitivi ed i tipi introdotti mediante la dichiarazione di una classe. Gli oggetti di una classe sono i valori del tipo. I valori dei tipi primitivi esistono, gli oggetti devono essere costruiti.
Per ogni tipo introdotto abbiamo considerato la dichiarazione di variabile e l'assegnamento di un valore ad una variabile:

```
boolean b = true;
char c = 'c';
int i = 10;
Punto p = new Punto (7,7);
```

C'è un altro tipo da considerare ed è il tipo **array di**.
Per esempio se, dati gli interi 12, 24, 36 costruiamo la tripla {12, 24, 36}, essa è un array di interi.
Il tipo degli elementi di un array può essere o un tipo primitivo o un qualsivoglia tipo creato in precedenza, possiamo pertanto parlare di array di interi, array di caratteri,…, array di oggetti.
Un array contiene un certo numero di valori, elementi dello stesso tipo. Il numero di elementi può essere 0, in tal caso si dice che l'array è vuoto. Se un array ha `n` elementi si dice che ha lunghezza `n`. Gli elementi di un array di lunghezza `n` sono indicate da un indice `0, …, n-1`.
Su un array sono definite le operazioni di estrazione di un elemento e di assegnazione di un valore ad un elemento; entrambe le operazioni richiedono l'accesso ad un elemento.

Il tipo di un array è costituito dal nome del tipo degli elementi che costituiscono l'array seguito da una coppia di parentesi quadrate (aperte e chiuse); pertanto se le componenti di un array sono di tipo TypeName, il tipo dell'array è **`TypeName[]`** (es: `int[], boolean[]`)

Dichiarazione di variabilie di tipo "array di" ed assegnamento

La dichiarazione:

```
int[] x;
```

significa che la variabile `x` è dichiarata di tipo `int[]`; essa non denota alcun oggetto, ad essa è assegnato il valore `null`. In modo analogo si può dichiarare:

```
long[] x;
Punto[] x;
Segmento[] x;
```

Poiché gli array sono oggetti, essi possono essere costruiti utilizzando l'operatore `new` seguito dal nome del tipo:

```
int[]x = new int[5];
Punto[] punti = new Punto[10];
```

Il primo statement significa che viene costruito un array di `5` componenti intere, l'array è denotato dalla variabile `x` ed ogni componente viene inizializzata a `0`. Il secondo statement significa che viene costruito un array di `10` componenti di tipo `Punto`, l'array è denotato dalla variabile `punti` ed ogni componente viene inizializzata a `null.`
Una volta creato un array, la sua lunghezza non cambia.

Gli elementi di un array possono essere inizializzati utilizzando una lista di espressioni, separate da virgole, racchiusa tra parentesi graffe. Ogni espressione specifica il valore di un elemento e deve essere dello stesso tipo dell'elemento. Il numero di espressioni deve essere uguale alla lunghezza dell'array. In altre parole, un valore di tipo "array di" può essere descritto "letteralmente" ponendo termini del tipo adeguato, separati da virgole, tra parentesi graffe. Se `x` denota l'array di 5 elementi di tipo `int[]`sopra costruito, si può assegnare ad `x` una quintupla di valori interi:

```
x = {12,4,7,24,36};
```

E' possibile effettuare la dichiarazione della variabile, la creazione dell'array e l'inizializzazione degli elementi in un unico statement:

```
int[] x = new int[] {12,4,7,24,36};
```

in questo caso indicare la lunghezza è opzionale.
La creazione di un array può essere fatta anche senza invocare l'operatore `new` ma assegnando direttamente ad una variabile di tipo array un valore del tipo corretto:

```
int[] x ={12,4,7,24,36};
```

Accesso agli elementi di un array

L'accesso alle componenti di un array avviene mediante l'operatore:

```
unArray[unIntero]
```

dove unIntero sta per un'espressione il cui risultato è un valore intero. Se `x` è la variabile di tipo `int[]` descritta sopra possiamo scrivere:

```
int i = x[0];   //i denota il valore 12
int j = x[1];   //j denota il valore 4
 x[3]=10;       //assegna il valore 10 al quarto elemento di x
```

Dopo l'assegnamento `x[3]=10` la variabile `x` denota la quintupla `{12,4,7,10,36}.`
Dato un array `x`, il termine `x.length` è di tipo `int` e denota la lunghezza della'array. Si può pertanto scrivere:

```
int i = x.length;
```

Molto spesso la inizializzazione di un array è fatta mediante un ciclo `for`:

```
int[]x = new int[100];
for (int i =0; i<x.length; i++)
   x[i] = 2*i;
```

Array multidimensionali

Abbiamo detto che gli elementi di un array possono essere di qualsiasi tipo, essi posssono pertanto essere anche di tipo array di. Per questa ragione un array multidimnsionale non deve necessariamente avere componenti della stessa lunghezza. Lo statement

```
int[][] matrix;
```

dichiara una variabile di tipo array di array di `int`. Lo statement

```
int[][] matrix = {{12,4},{7,10},{36,1}};
```

crea una array di tre componenti ognuno dei quali è un array di `int` di lunghezza due.
Lo statement

```
int[][] matrix = new int[3][3];
```

è equivalente al codice

```
int[][] matrix = new int[3][];
for(int i=0; i<matrix.length; i++)
   matrix[i] = new int[3];
```

Un array bidimensionale può essere inizializzato nel modo seguente:

```
int[][] numbers;
```

```
numbers = new int[10][2];
for(int i=0; i<numers.length; i++) {
  for(int j=0; j<numers[i].length; j++) {
    numbers[i][j]=2*i+1;
  }
}
```

Una matrice triangolare può essere creata facilmente:

```
float[][] triang = new float[100][];
for(int i=0; i<triang.length; i++)
  triang[i] = new float[i+1];
```

Quando si creano array multidimensionali è richiesto che almeno la lunghezza del primo array sia diversa da 0, es:

```
int[][][][][] multArr = new int[3][][][][];
```

Una classe poligono

Definiamo il seguente modello di poligono:

1. Un poligono possiede una n-pla di vertici;
2. Un poligono fornisce il vertice i ($0 \leq i \leq n-1$);
3. Un poligono imposta il il vertice i ($0 \leq i \leq n-1$);
4. Un poligono fornisce il suo lato i (per i<n-1, il lato i è il segmento che congiunge il vertice i al vertice i+1; il lato n-1 è il lato che congiunge il vertice n-1 al vertice 0);
5. Un poligono è capace di traslarsi;
6. Un poligono è in grado di verificare se è uguale o diverso da un altro poligono;
7. Un poligono calcola la sua lunghezza.

Dichiariamo ora una classe, il cui nome è `Poligono`, che realizza il modello sopra definito.
La classe avrà:

- due variabili private, una di tipo `int` per rappresentare il numero di vertici del poligono ed una di tipo `Punto[]` per rappresentare la n-upla di vertici;
- un costruttore pubblico che prende in ingresso un array di punti;
- i metodi pubblici:

```
public int getVertexCount()
public Punto getVertex(int i) {…}
public void getVertex(int i, Punto p) {…}
public Segmento getEdge(int i) {…}
public void translateOf(double dx, double dy){…}
public boolean equals(Poligono pol) {…}
public double perimeter() {…}
```

Ad un oggetto di tipo `Poligono` può essere richiesta l'esecuzione di qualsiasi metodo pubblico della classe.
Gli oggetti che devono interagire con oggetti di tipo `Poligono` devono conoscere unicamente l'intestazione dei metodi pubblici della classe, l'interfaccia della classe.

```
public class Poligono {

  // variabile privata
  private Punto[] vertici;

  // costruttore pubblico
  public Poligono (Punto[] punti){
    if(punti!=null){
      vertici = new Punto[punti.length];
      for(int i=0; i<punti.length; i++){
        if(punti[i]==null)System.exit(0); //se ci sono punti nulli esci
        else vertici[i]= new Punto(punti[i].getX(),punti[i].getY());
      }
    }
  }
```

```
    // metodi pubblici
     public int getVertexCount() {
       return vertici.length;
     }
     public Punto getVertex(int i) {
       return vertici[i];
     }
     public void setVertex(int i, Punto p){
       if(p!=null) vertici[i]=p;
     }
     public Segmento getEdge(int i){
       int n=vertici.length;
       if(i<0 || i>n-1) return null;
       if(i<n-1) return new Segmento(vertici[i], vertici[i+1]);
       else return new Segmento(vertici[n-1], vertici[0]);
     }
     public void translateOf(double dx, double dy) {
       for(int i=0; i<vertici.length; i++)
         vertici[i].translateOf(dx, dy);
     }
     public boolean equals(Poligono pol){
       if(pol==null)return false;
       for(int i=0; i<vertici.length; i++) {
         Punto p1 = vertici[i];
         Punto p2 = pol.getVertex(i);
         if (!p1.equals(p2)) return false;
         // if(!vertici[i].equals(pol.getVertex(i))) return false;
       }
       return true;
     }
     public double perimeter (){
       double len=0.0;
       for(int i=0; i<vertici.length; i++)
         len=len+getEdge(i).length();
     }
  }
```

Il costruttore impedisce di costruire un poligono i cui vertici siano punti nulli; il modo in cui lo impedisce, mediante lo statement `System.exit(0)` che interrompe l'esecuzione, non è corretto. Vedremo come trattare correttamente situazioni come questa quando considereremo il trattamento delle eccezioni.
Per costruire l'array `vertici` abbiamo costruito un nuovo array di nuovi punti e copiato in ognuno di essi l'elemento corrispondente dell'array `punti`, fornito dall'esterno. Avremmo potuto effettuare l'assegnamento dell'oggetto esterno alla nostra variabile:

```
vertici = punti;
```

ma questo espone lo stato del nostro poligono a modifiche esterne non controllabili.
Poiché anche gli elementi di tipo `Punto` sono modificabili, è opportuno generare nuovi punti da assegnare alle variabili `vertici[i]`. Lo statement:

```
vertici[i]=punti[i]
```

produrrebbe infatti una modifica del poligono se uno degli elementi dell'array `punti` venisse modificato.

Poiché abbiamo impedito la costruzione di un poligono con punti nulli, i metodi della classe non devono preoccuparsi di verificare se vertici e lati sono nulli.

La classe `String`

In Java le stringhe sono oggetti; esse sono esemplari di una classe pubblica, la classe **`String,`** che realizza un modello di cui elenchiamo le caratteristiche:

1. una stringa è una sequenza, eventualmente vuota, di caratteri di un alfabeto (Unicode);
2. una stringa ha una lunghezza (`n`) ;
3. una stringa è in grado di verificare se è uguale o diversa da una stringa data, anche ignorando se i caratteri sono maiuscoli o minuscoli;
4. è in grado di verificare se una stringa data è uguale ad una sua sottostringa;
5. una stringa è in grado di fornire informazioni sui suoi elementi:
   il suo carattere in posizione `i` (`0≤ i ≤n`) ;
   tutti i suoi caratteri
   la posizione della prima e ultima occorrenza di un suo carattere a partire da un indice fissato

Nella classe ci sono inoltre metodi per generare un'altra stringa (factory methods) trasformando la stringa data, e metodi statici che forniscono la rappresentazione mediante una stringa dei valori dei tipi primitivi.

Un oggetto di tipo `String`, una volta creato, è **immutabile** e rappresenta una sequenza di lunghezza fissata di caratteri Unicode. Una stringa (oggetto di tipo `String`) è rappresentata da uno *string literal*, costituito dalla sequenza di caratteri che costituiscono l'oggetto rappresenato, racchiusi tra doppie virgolette. Esempi di string literal:

"" // la stringa vuota
"\"" // la stringa costituita dal solo carattere "
"abcd" // la stringa costituita dai 4 caratteri a, b, c, d
"oggi è una brutta giornata" // la stringa costituita da 26 caratteri, quelli delle 5 parole e gli spazi bianche tra esse.

Una stringa si costruisce utilizzando uno dei 7 costruttori della classe, ora ci limitiamo a considerarne quattro:

```
public String(){};
public String(String value){};
public String(char[] data){};
public String(char[], int offset, int count){};
```

Il primo costruttore costruisce la stringa vuota; il secondo costruisce la stringa corrispondente al letterale ottenuto in ingresso; il terzo costruisce una stringa da un array di caratteri; il quarto costruisce una stringa da un sotto-array di un array di caratteri, per precisare il sotto-array sono forniti gli interi `offset e count` che rappresentano ripettivamente l'indice del primo carattere e la lunghezza del sotto-array da considerare.

Dichiarazione di variabile e assegnamento

```
String s = new String();
String s = new String ("oggi è una brutta giornata");
char[] caratteri = {'a', 'b', 'c', 'd'};
String s = new String (caratteri);
String s = new String (caratteri, 1, 3);
```

è anche possibile scrivere:

```
String s = "oggi è una brutta giornata";
String s = "";
```

nei due esempi sopra, è chiamato implicitamente il costruttore che ha come argomento una stringa.

Metodi

Ogni stringa può eseguire le operazioni del modello sopra descritto; esse sono realizzate dai seguenti metodi pubblici:

```
public int length(){};
public String toString(){};
public int compareTo(String anotherString){};
public boolean equalsIgnoreCase(String anotherString){};
public char charAt(int index){};
```

```
public boolean startWith(String prefix){};
public boolean startWith(String prefix, int offset){};
public boolean endWith(String suffix){};
public int indexOf(int ch){}; //se ch non è contenuto nella stringa ritorna -1
public int indexOf(int ch, int fromIndex){};
public int lastIndexOf(int ch){};
public int lastIndexOf(int ch, int fromIndex){};
public char[] toCharArray(){};
public void getChars(int srcBegin,int srcEnd,char[] dst,int dstBegin){};
public String substring(int beginIndex){};
public String substring(int beginIndex, int endIndex){};
public String toLowerCase(){};
public String toUpperCase(){};
public String replace(char oldChar, char newChar){};
public String concat(String str)
public String toString()
```

Dei metodi che rendono una stringa, i primi sei costruiscono una nuova stringa invocando uno dei costruttori della classe, il metodo `toString` invece, rende la rappresentazione della stringa.
L' esecuzione di ognuno dei metodi sopra elencati può essere richiesta ad ogni stringa; es:

```
String s = "il mio gatto è simpatico";
int lun=s.length();
boolean confronto = s.compareTo("luna");
boolean con = s.equalsIgnoreCase("il Mio Gatto è Simpatico");
char c = s.charAt(5);
boolean b = s.startsWith("il");
boolean b1 = s.startsWith("il", 5);
boolean b2 = s.endsWith("tico");
int i =s.indexOf(a);
int j =s.indexOf(a,3);
int l =s.lastIndexOf(a);
int la =s.lastIndexOf(a,3);
char[] caratteri = s.toCharArray();
char[] dest = s.getChars(0, s.length-1, dest, 0);
String sub = s.substring(5);
String sub1 = s.substring(5,10);
String s1=s.toLowerCase();
String s2=s.toUpperCase();
String s3=s.replace('o','a');
String aString = "e bello";
String str =s.concatenate(aString);
```

La classe `String` contiene anche metodi pubblici `static`; essi vengono richiesti alla classe e non ad un suo oggetto. Alcuni di essi, che riportiamo qui, forniscono la rappresentazione mediante una stringa, dei valori dei tipi `boolean, char, int, long, double`:

```
public static String valueOf(boolean b){};
public static String valueOf(char c){};
public static String valueOf(int i){};
public static String valueOf(long l){};
public static String valueOf(float f){};
public static String valueOf(double d){};
```

Quando si esegue un programma è necessario visualizzare (stampare) il valore dei risultati. A tal fine, si invoca di frequente il metodo pubblico `println(String str)`di una classe, `System`, i cui componenti sono in grado di stampare. La variabile `out` rappresenta un dispositivo di output della classe `System`.
Lo statement:

```
System.out.println(s)
```

produce la visualizzazione della stringa `s`, cioè:

```
il mio gatto è simpatico
```

Poiché è necessario visualizzare (stampare) valori di qualunque tipo, è necessario che essi siano trasformati in stringhe. Per i valori dei tipi primitivi ciò è fatto automaticamente; se `x` è una variabile di uno dei tipi primitivi quando si scrive:

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

viene stampato il letterale che rappresenta il valore denotato da `x` (gli `int` e i `long` sono rappresentatati in notazione decimale).
Per i tipi da noi creati (`Punto, Segmento, Poligono`,…) è necessario fornire ogni classe di un metodo pubblico, che chiamiamo `toString,` la cui esecuzione da parte di un oggetto della classe produce come risultato una stringa che rappresenta l'oggetto.
Per esempio, un metodo `toString` della classe `Punto` potrebbe visualizzare la stringa:
" punto p: (x = valore di x, y = valore di y)"

In essa le parti sottolineate rappresentano i valori delle variabili che per essere stampati devono essere trasformati in stringhe. La stringa che si ottiene è pertanto dipendente dai valori delle variabili, non è una stringa non modificabile e quindi non è un oggetto di tipo String.
Per costruire metodi che consentono di rappresentare un oggetto mediante una stringa di caratteri dobbiamo introdurre un' altra classe.

La classe `StringBuffer`

Se una stringa è un oggetto immutabile, la classe `StringBuffer` realizza il modello di una sequenza di caratteri modificabile per lunghezza e contenuto. Il modello di `StringBuffer` estende il modello di `String`:

1. la sequenza di caratteri ha una lunghezza che può variare a causa di:
   concatenazione di una sequenza
   inserzione di una sequenza in qualsiasi posizione del buffer
   cancellazione di una sottosequenza in qualsiasi posizione del buffer

2. la sequenza può fornire informazioni sul contenuto
3. la sequenza può modificare un carattere (non può farlo se non generando un'altra stringa)

Le operazioni descritte informalmente nei punti sopra elencati sono realizzate dai seguenti metodi pubblici:

```
public int capacity(){};
public void ensureCapacity(int minCapacity){};
public int length(){};
public int setLength(){};
public String toString(){};
public char charAt(){};
public void setCharAt(int index, char ch){};
public void getChars(int srcBegin, int srcEnd){};
public StringBuffer append(String str){};
public StringBuffer append(char[]  str){};
public StringBuffer append(char[]  str, int offset, int len){};
public StringBuffer append(boolean b){};
public StringBuffer append(char c){};
public StringBuffer append(int i){};
public StringBuffer append(long l){};
public StringBuffer append(float f){};
public StringBuffer append(double d){};
public StringBuffer insert(int offset, String str);
public StringBuffer insert(int offset, char[]  str);
public StringBuffer insert(int offset, boolean b);
public StringBuffer insert(int offset, char c);
public StringBuffer insert(int offset, int i);
public StringBuffer insert(int offset, long l);
public StringBuffer insert(int offset, float f);
public StringBuffer insert(int offset, double d);
public StringBuffer reverse();
```

I metodi più importanti della classe `StringBuffer` sono i metodi pubblici `append` e `insert`; essi sono overloaded in modo da accettare dati di tipi diversi.
Tutti i metodi `append` modificano il contenuto di uno `StringBuffer` concatenando alla stringa contenuta in esso, una stringa, i caratteri di un array, la stringa che rappresenta un valore di uno dei tipi primitivi. I valori dei tipi primitivi vengono automaticamente convertiti in una stringa prima di essere concatenati con la stringa contenuta nel buffer, ciò è

fatto mediante i metodi statici `valueOf` della classe `String`. Un'operazione di `append` modifica la lunghezza dello `StringBuffer`, se la lunghezza della stringa contenuta supera la capacità questa viene automaticamente aumentata. Anche i metodi `insert` modificano la lunghezza dello `StringBuffer`, anche in questo caso la sua capacità viene automaticamente aumentata se necessario.
Il metodo `reverse` non produce nessuna modifica della lunghezza e della capacità dello `StringBuffer` tuttavia la stringa che esso contiene è diversa.

La classe ha una variabile privata di tipo `char[]` ed una variabile privata di tipo `int`; la prima è utilizzata per memorizzare i caratteri la seconda per contare il numero dei caratteri memorizzari nell'array. Quando è necessario aumentare la capacità viene generato un nuovo array di capacità adeguata.
La classe ha tre costruttori pubblici:

```
public StringBuffer(){};
public StringBuffer(int length){};
public StringBuffer(String str){};
```

Il costruttore privo di parametri costruisce uno `StringBuffer` di capacità fissata (pari a 16 caratteri), contenente la una sequenza di caratteri vuota.
Il costruttore che ha come parametro un valore intero, costruisce uno `StringBuffer` di capacità pari a quella precisata dal valore del parametro; lo `StringBuffer` contiene una sequenza di caratteri vuota.
Il costruttore che ha come parametro un valore di tipo String costruisce uno `StringBuffer` di capacità pari alla lunghezza della stringa in ingresso più la capacità fissata (pari a 16 caratteri); lo `StringBuffer` contiene la stringa in ingresso.

Avendo a disposizione la classe `StringBuffer` possiamo definire un metodo `toString` per la classe Punto. Potrebbe essere:

```
public String toString (){
  StringBuffer buf=new StringBuffer("punto p:[x = ");
    buf.append(x);
    buf.append(", y = ");
    buf.append(y);
    buf.append(']');
    return buf.toString();
}
```

C'è un altro modo per scrivere tutto ciò:

```
public String toString (){
  String s=" punto p:[x = " + x + ", y = " + y +"]";
  return s;
}
```

Il simbolo di operazione + è utilizzato per denotare la concatenazione tra stringhe;
Il codice:

```
String x = "a" +4+ "c"
```

è equivalente a:

```
String x = new StringBuffer().append("a").append(4).append("c").toString();
```

Ereditarietà

Abbiamo considerato la dichiarazione di una classe, mostrato come crearne esemplari e come richiedere ad essi di eseguire le operazioni specificate dai metodi della classe.

Si possono dichiarare nuove classi come sottoclassi di classi esistenti. Si parla di **ereditarietà** tra classi.
Illustriamo il concetto di ereditarietà considerando il mondo degli esseri viventi, poiché in essi l'ereditarietà è realizzata di fatto. Quando affermiamo che un animale o un vegetale sono essere viventi, intendiamo dire che li accumunano alcune caratteristiche che determinano la capacità di vivere. Tuttavia, la gran parte degli elementi essenziali per la loro vita è realizzata in essi in modo differente. Il tipo essere vivente è un'astrazione: non esiste nessun esemplare di essere vivente che non appartenga al regno animale o vegetale. Se poi consideriamo il tipo animale e il tipo vegetale ci accorgiamo che anch'essi sono un'astrazione, i loro esemplari sono a loro volta classificabili in articolate gerarchie di phyla, classi, sottoclassi, specie,… in funzione di ciò che li accomuna e li distingue.
Gli esemplari delle diverse sottoclassi di una classe hanno alcune caratteristiche comuni, per esempio i mammiferi hanno in comune con gli altri vertebrati (pesci, anfibi, rettili, uccelli) la presenza della colonna vertebrale; potremmo dire che la classe vertebrati è caratterizzata dalla colonna vertebrale e che le sue sottoclassi la ereditano, è nel loro patrimonio genetico.

Come nel mondo biologico si ereditano caratteristiche (strutture e funzioni), così nei linguaggi ad oggetti si possono dichiarare classi che ereditano da classi già costruite. Le classi che ereditano sono **sottoclassi dirette** di una classe, la classe da cui si eredita è detta **superclasse diretta**. Il consentire l'ereditarietà evita di specificare ogni volta ciò che è comune ad esemplari di classi diverse. La classe che eredita aggiunge alle caratteristiche della superclasse caratteristiche che la distinguono, pertanto estende la superclasse. Es:

```
public abstract class EssereVivente{}
public abstract class Vegetale extends EssereVivente{}
public abstract class Animale extends EssereVivente{
    public abstract void mangia (EssereVivente cibo){}
    public abstract String parla() {}
}
public class Leone extends Animale {…
  public void mangia (EssereVivente cibo){
     if (cibo == null || or !(cibo instnceof Animale)) return;
        …
  }
  public String parla() {
        return "rrrr rrrr";
  }
}
public class Pecora extends Animale {…
  public void mangia (EssereVivente cibo){
    if (cibo == null || or !(cibo instnceof Vegetale)) return;
     …
  }
  public String parla() {
        return "beeh beeh";
  }
}
```

Abbiamo dichiarato cinque classi, tre di esse `EssereVivente`, `Vegetale` e `Animale` sono **classi astratte**; ciò significa che non si possono costruire esemplari della classe. Nel nostro esempio ha senso costruire particolari animali o vegetali ma non ha alcun senso costruire un generico animale o vegetale.
Una classe, per essere dichiarata astratta, deve contenere almeno un metodo astratto, cioè privo di corpo e quindi non eseguibile. Classi e metodi sono dichiarati astratti anteponendo al loro nome la parola chiave `abstract`.
Le classi astratte possono avere costruttori; tali costruttori possono essere invocati solo dai costruttori delle sottoclassi non astratte.
La definizione:

```
class X extends Y{…}
```

significa che la classe X eredita tutti i componenti (variabili e metodi) della classe Y.
Cosa vuol dire ereditare i componenti? Vuol dire farli propri senza dichiararli di nuovo.

Consideriamo l'ereditarietà dei componenti pubblici di istanza. La classe X non deve dichiararli di nuovo, essa ha a disposizione tutto il codice già scritto della classe Y, ogni esemplare di tipo X ha accesso diretto ad essi.
Ed i componenti privati di istanza? Essi sono ereditati ma non sono accessibili, un oggetto della classe X non ha accesso a variabili e metodi privati della classe Y. Ereditarietà ed accessibilità sono concetti distinti.

Consideriamo un esempio; se in Y sono dichiarati:

```
public int anInt;
public int aMethod(){…}
private int myInt;
private int myMethod(){…}
```

nei metodi di X si potranno scrivere statement quali:

```
anInt=5;  //la variabile dichiarata in Y è accessibile
int i =x.aMethod(); //il metodo è accessibile in  X
```

Al contrario non si potranno scrivere statement quali:

```
myInt=5;  // myInt è accessibile solo in Y
int i =x.myMethod();// myMethod è accessibile solo in  Y
```

In che senso allora le componenti private `myInt` e `myMethod()` sono ereditate? Nel senso che quando viene costruito un oggetto di tipo X è come se venisse costruito un oggetto di tipo Y a cui si aggiungono poi tutte le nuove componenti di X. Se torniamo all'esempio precedente ciò equivale a dire che:

- dentro ogni esemplare di tipo `Leone` c'è un esemplare di tipo `Animale` (anche con le sue componenti private non accessibili ad un leone)
- dentro un `Animale` c'e un `EssereVivente` (anche con le sue componenti private non accessibili ad un animale).

Tornando al parallelo biologico, ciò equivale a riconoscere che in ogni mammifero c'è un uccello, in esso un rettile, in esso un anfibio, in esso un pesce ma le caratteristiche degli antenati non sono accessibili (utilizzabili) al mammifero.

Overriding di metodi

I metodi ereditati, se pubblici, possono essere ridichiarati.
Se in una sottoclasse si dichiara un metodo di istanza con la stessa segnatura di un metodo di istanza della superclasse si dice che il metodo della sottoclasse ricopre, **override**, il metodo della superclasse con la stessa segnatura. E' necessario che i due metodi abbiano lo stesso return type.
Se il metodo dichiarato nella sottoclasse ricopre un metodo astratto della superclasse si dice non solo che lo ricopre ma che lo implementa (i metodi astratti sono privi di corpo). Nel nostro esempio i metodi `parla` e `mangia` implementano i corrispondenti metodi astratti della classe `Animale`. Se l'esecuzione del metodo `parla` viene richiesta ad un leone esso produrrà un ruggito, se viene richiesta ad una pecora, essa produrrà un belato.
La dichiarazione di un metodo astratto in una classe determina un obbligo per le classi eredi che non sono a loro volta classi astratte: esse devono implementarlo. Pertanto, un metodo dichiarato `abstract` non può essere dichiarato `private.`
Un metodo della superclasse diretta, anche se ricoperto, è accessibile nella sottoclasse. Se `M` è il nome del metodo della sottoclasse che ricopre il metodo `M` con la stessa segnatura, il termine `super.M.` denota il metodo ricoperto della superclasse. Esempio:

```
public class Y{…
  public int anInt;
  public int aMethod(){…}
…
}

public class X extends Y{…
  public int aMethod(){
    int i=super.aMethod(); //viene invocato il metodo ricoperto di Y
```

```
    …
    }
    public double aNewMethod(){
    }
  …
}
```

Nell'esempio sopra, il metodo `aMethod` di `Y` viene ricoperto ed esteso; ricoperto perché in `X` viene dichiarato un metodo con lo stesso nome e segnatura di un metodo di `Y`, esteso perché nel corpo di `aMethod` di `X` si invoca il metodo ricoperto e si aggiungono statement.
Questo è uno degli usi più frequenti dell'overriding, la sottoclasse vuole eseguire concettualmente lo stesso compito ma per farlo deve aggiungere ulteriori azioni a quelle della classe da cui eredita.

E' possibile vietare l'overriding di un metodo dichiarandolo finale mediante la parola chiave **`final`**; es:

```
public class Y{…
  public final double aFinalMethod(){…}
…
}
```

E' opportuno dichiarare `final` metodi essenziali la cui ricopertura potrebbe determinare comportamenti inconsistenti.

Hiding di variabili

Anche le variabili pubbliche possono essere ridichiarate; se ridichiarata una variabile nasconde (**hide**) la variabile della superclasse con lo stesso nome, la variabile della superclasse risulta comunque accessibile utilizzando la parola chiave `super`. Tuttavia, la ridichiarazione di una variabile pubblica della superclasse è da evitare; non c'è alcuna necessità di utilizzare per una nuova variabile lo stesso identificatore di una variabile ereditata dalla superclasse.
Al contrario ricoprire un metodo è spesso utile e non crea alcuna ambiguità; il nuovo metodo ha infatti la stessa segnatura ma un corpo diverso, precisa meglio le azioni che gli esemplari della sottoclasse devono svolgere.

Eredità di componenti private

Abbiamo definito le condizioni in cui possono essere ridichiarate le componenti pubbliche ereditate da una superclasse, e le componenti private? Poiché esse non sono accessibili possono essere ridichiarate senza alcun problema; non c'è alcun legame tra esse e le componenti private della superclasse che hanno lo stesso nome. Es:

```
public class Y{…
  private int anInt;
  private int aMethod(){…}
…
}

public class X extends Y{…
  private int anInt;
  private double aMethod(){…};
  }
 …
}
```

Accesso riservato agli eredi

Può essere importante rendere alcune componenti di una classe accessibili solo alle sottoclassi e private per tutte le altre classi. Per realizzare ciò si utilizza la parola chiave **`protected`**. Una componente dichiarata `protected` è accessibile a tutte e sole le sottoclassi di una classe, per esse è come se la componente fosse dichiarata pubblica, per le altre classi è come fosse dichiarata privata.
E'necessario quando si ricopre un metodo tenere conto anche dell'accessibilità: un metodo pubblico può essere ricoperto solo da un metodo pubblico, un metodo protetto può essere ricoperto solo da un metodo protetto o pubblico.
Per i metodi privati non ci sono problemi, di fatto essi non vengono ricoperti anche se si dichiara un metodo con la stessa segnatura di fatto si dichiara un'altra cosa.
L'utilizzo sapiente degli attributi di accesso è un problema di architettura.

Per le componenti dichiarate `static` l'ereditarietà non funziona; se si vuole un metodo ereditabile e ricopribile bisogna disegnarlo come metodo di istanza. Un metodo dichiarato `abstract` non può essere dichiarato anche `static`.

Abbiamo parlato dell'ereditarietà dei componenti, ed i costruttori? **I costruttori non si ereditano**. Una sottoclasse invoca il costruttore della superclasse diretta mediante la parola chiave `super` seguita dagli argomenti opportuni.

Riepilogo
Se `X` è una sottoclasse diretta di Y:

- tutte le variabili private di `Y` vengono ereditate ma non sono accessibili da X;
- tutti i metodi privati di `Y` vengono ereditati ma non sono accessibili da X;
- tutte le variabili pubbliche e protette di `Y` vengono ereditate e sono accessibili da X;
- tutti i metodi pubblici e protetti di `Y` vengono ereditati e sono accessibili da X.

X non può ricoprire i metodi dichiarati `final` in Y.

X può ridichiarare:

- una variabile privata con lo stesso identificatore di una variabile di `Y`, di fatto è una variabile non collegata a quella, non accessibile, ereditata dalla superclasse diretta;
- un metodo privato con lo stesso nome e segnatura di un metodo di `Y`; di fatto i due metodi non sono collegati e possono pertanto avere diversi return type;
- una variabile pubblica o protetta con tipo diverso di una variabile di `Y`, la variabile di `Y` è accessibile utilizzando `super.varIdentifier`. Anche se il compilatore consente ciò, è buona norma evitare di ridichiarare una variabile accessibile di una superclasse;
- un metodo pubblico o protetto con la stessa segnatura di un metodo ereditato; Il nuovo metodo ricopre quello ereditato, deve pertanto avere la stessa o più ampia accessibilità; il metodo ricoperto è accessibile nella sottoclasse diretta attraverso `super.methodName()`.

Osservazione
La dichiarazione di una sottoclasse consente di arricchire (estendere) il comportamento degli esemplari della superclasse diretta aggiungendo nuovi metodi e arricchendo l'implementazione dei metodi che vengono ricoperti.
Il meccanismo dell'ereditarietà consente di utilizzare codice già scritto e aiuta l'astrazione:

- consente di utilizzare per la sottoclasse il codice già scritto per la superclasse;
- aiuta l'astrazione poiché in una superclasse possono essere dichiarati tutti i metodi comuni alle sottoclassi; alcuni di essi possono essere dichiarati astratti, di altri si può fornire un'implementazione. Un metodo dichiarato `abstract` non può essere dichiarato né `private` né `static`.
- Una classe in cui sono dichiarati metodi astratti deve essere dichiarata astratta. Ogni sottoclasse non astratta di una classe astratta deve ridichiarare i metodi astratti fornendo per essi un'implementazione; inoltre può ridichiarare i metodi accessibili (pubblici e protetti) ereditati. E' opportuno dichiarare una classe astratta quando si vuole forzare l'implementazione di tutti i metodi che le sottoclassi dirette ereditano e lasciarne altri solo dichiarati in modo che ogni sottoclasse fornisca la propria implementazione.

Sottoclassi dirette e sottoclassi

Per chiusura transitiva della relazione è sottoclasse diretta di si ottiene la realzione è sottoclasse di. Si dice che una classe S è sottoclasse di una classe T se vale una delle seguenti condizioni:

- S è una sottoclasse diretta di T
- Esiste una classe C tale che S è sottoclasse di C e C è sottoclasse di T, applicando la definizione ricorsivamente.

Una classe non può essere dichiarata sottoclasse di se stessa, pertanto definizioni circolari come quella che segue non sono ammesse:

```
class S extends class C
class C extends class S
```

Si dice che C è una superclasse (superclass o ancestor class) di S se S è una sottoclasse di C. Quindi con la parola superclasse si intende il genitore diretto o qualsiasi antenato.

Abbiamo detto che una classe definisce un tipo. Se la classe Leone eredita dalla classe Animale, un esemplare di Leone è di tipo Leone ed anche di tipo Animale. Pertanto, se T ed S sono classi, e S è sottoclasse di T, gli esemplari di S possono essere dichiarati di tipo T, e gli esemplari di T possono essere forzati al tipo S a patto che siano anche di tipo S. Es:

```
Animale  a =  unLeone;
Leone l = (Leone) a;    //OK in esecuzione
Pecora p = (Pecora) a;  //produrrà un errore in esecuzione
```

Per evitare errori di esecuzione è sempre opportuno controllare che l'oggetto sia un esemplare del tipo a cui lo si vuole convertire. Si può fare questo mediante l'operatore **`instanceof.`**
L'operatore `instanceof` consente di determinare se un oggetto è di un certo tipo. E' un operatore che fornisce un valore di tipo `boolean`. Es:

```
if (a instanceof Leone) l = (Leone)a;
```

Una volta istanziata una variabile `x` di tipo T, si può utilizzare `x` per richiedere tutti e soli i metodi previsti dal tipo T.

Considerando i metodi protetti di una catena di classi si può capire meglio cosa significa asserire che una classe `X` eredita anche l'informazione privata dalle superclassi. Infatti può accadere che nel corpo di qualcuno dei metodi di `X` vengano invocati metodi protetti della superclasse diretta di `X`, che in questi siano invocati metodi protetti della propria superclasse diretta, che in questi siano invocati metodi protetti della propria superclasse diretta…ed ognuno di essi può operare su variabili private o utilizzare metodi privati della propria classe.
In questo modo, quando un esemplare della classe `X` esegue uno dei metodi di `X` con le caratteristiche sopra citate, esso in realtà esegue metodi delle superclassi senza invocarli direttamente (li ha ereditati, li ha dentro)**.**

Classi `final`

L'ereditarietà è uno strumento potentissimo, essa consente a chiunque di estendere una classe pubblica aggiungendo ad essa nuovi metodi e ricoprendo metodi esistenti. Per fare ciò non è necessario conoscere l'implementazione dei metodi, è sufficiente conoscerne la dichiarazione. Se aggiungere nuovi metodi non pone in generale problemi, la ricopertura di un metodo potrebbe determinare comportamenti non consistenti con il comportamento atteso. Chiariamo il concetto con un esempio.
Supponete che io estenda la classe `String` aggiungendo nuovi metodi e ricoprendo il metodo `length` in modo che esso non fornisca la lunghezza di una stringa ma un valore diverso (un valore random, il doppio della lunghezza,…); potrei anche, inconsapevolmente, fare un errore mentre scrivo il codice. Se qualcuno usasse la mia classe costruendone dei suoi esemplari, otterrebbe degli oggetti che si comportano come esemplari di `String` eccetto che nell'esecuzione del metodo `length` ma questo potrebbe rivelarsi catastrofico.
Ciò potrebbe accadere per qualsivoglia classe di oggetti ben conosciuti, qualcuno estende la classe, rende i suoi esemplari disponibili in rete, chi li usa suppone di usare oggetti con comportamenti stabiliti (pattuiti) da un modello ed invece ha comportamenti diversi.

Per evitare problemi di questo tipo, metodi particolarmente importanti vengono dichiarati `final`, non ricopribili. Anche intere classi vengono dichiarate `final`, non estensibili; di una classe dichiarata `final` non si possono definire sottoclassi.

Le classi `String` e `String Buffer` sono dichiarate `final`, anche il tipo `[]` (array di) è `final`; stringhe ed array sono oggetti troppo importanti per consentire di modificarne i comportamenti. Altre classi essenziali in Java sono `final`.

La classe `Object`

C'è in Java una classe di cui tutte le classi sono sottoclassi; tale classe si chiama `Object` ed è la radice unica della gerarchia delle classi Java. Ciò significa che se una classe non utilizza la parola chiave `extends` ha come superclasse diretta, implicita, la classe `Object`.

La classe `Object` dichiara e implementa metodi che definiscono il comportamento di base di qualsiasi oggetto. Ogni oggetto, anche quelli di tipo array di, può eseguire i metodi dichiarati in `Object`.
Ad una variabile dichiarata di tipo `Object` si può assegnare qualunque oggetto, quindi anche un `array di`; non si può assegnare ad essa un valore di uno dei tipi primitivi.

```
Object obj; //obj è dichiarata di tipo Object, non denota niente
```

```
obj=34; // ERRORE, 34 non è un oggetto
obj={34,42}; //assegnamento corretto, un vettore di interi è un oggetto
int i= obj[0]; //ERRORE, un oggetto generico non ha componenti
int i = ((int[])obj)[0]; //OK, obj denota un array di int
char c=((char[])obj)[0]; //ERRORE in esecuzione
obj="sono una stringa" //OK
char c= ((String)obj).charAt(5); //OK
obj=new Punto(5,5); //OK
((Punto)obj).translateOf(24,12); //OK
```

Dei metodi della classe `Object`, alcuni possono, e spesso devono, essere ricoperti, altri non possono esserlo, sono `final`. Dei cinque metodi di `Object` che possono essere ricoperti, consideriamo i tre metodi pubblici che seguono:

```
public boolean equals(Object obj){}
public int hashCode(){}
public String toString(){}
```

Il metodo `equals`

Indica se un oggetto è uguale a questo oggetto (l'oggetto che esegue il metodo).
Il metodo `equals` deve implementare una relazione di equivalenza; chi implementa un metodo `equals` deve preoccuparsi di soddisfare queste condizioni:

- per ogni valore `x` di un tipo reference, `x.equals(x)` deve produrre come risultato `true` (riflessività);
- per ogni coppia `x`, `y` di valori di tipo reference, `x.equals(y)` deve produrre come risultato `true` se e solo se `y.equals(x)` produce come risultato `true` (simmetria);
- per ogni terna `x`, `y`, `z` di valori di tipo reference, se `x.equals(y)` produce come risultato `true` e `y.equals(z)` produce come risultato `true` allora `x.equals(z)` deve produrre come risultato `true` (transitività);
- per ogni valore `x` di un tipo reference, e `x` diverso da `null`, `x.equals(null)` deve produrre come risultato `false;`
- per ogni coppia `x`, `y` di valori di tipo reference, la invocazione ripetuta di `x.equals(y)` deve consistentemente produrre come risultato `true` oppure `false`, se non è stata modificata alcuna delle informazioni usate da `x` e `y` per la comparazione (consistenza).

Il metodo `equals` di `Object` implementa la più discriminativa tra le relazioni di equivalenza, l'identità; cioè per ogni coppia `x`, `y` di valori di tipo reference, `((Object)x).equals(y)` produce come risultato `true` se e solo se `x` e `y` denotano lo stesso oggetto `(x==y).`

Il metodo `equals` di `Object` deve essere ricoperto in ogni classe che viene costruita infatti, in generale, si è interessati a verificare se due termini denotano oggetti uguali (secondo criteri di uguaglianza adeguati per la classe) e non se essi denotano lo stesso oggetto.
Nella classe `String equals` è ricoperto. Nella classe `array` non è ricoperto, pertanto l'unica relazione di uguaglianza definita tra due "array di" è l'identità, es:

```
int[] x= new int[]{3,4,5};
int[] y= new int[]{3,4,5};
boolean confronto= x.equals(y);
```

la valutazione del metodo `equals` produce come risultato `false`; è come se avessimo scritto:

```
boolean confronto= x==y;
```

Come esempio di overriding, dichiariamo in `Punto` un metodo `equals` con la stessa segnatura del metodo `equals` di `Object` (il metodo di istanza che avevamo dichiarato aveva come argomento un punto):

```
public class Punto{
  …
  public boolean equals(Object obj){
    if(this==obj) return true;

    if(obj!= null && obj instanceof Punto) {
```

```
      Punto p =(Punto)obj;
      return(x==p.x && y==p.y);
    }
    return false;
…
}
```

Ora la nostra classe `Punto` ha tre metodi `equals`, uno dei quali è `static`. Essi hanno diverse segnature, si ha pertanto un overloading del nome `equals`. Poiché le segnature sono diverse, dovrebbe essere chiaro a quale metodo ci si riferisce quando se ne richiede la esecuzione; in realtà in questo caso può sorgere un'ambiguità. Infatti, quando ad oggetto di tipo `Punto` viene richiesta l'esecuzione di `equals` con un argomento di tipo `Punto` a quale dei due metodi `equals` ci si vuole riferire? Un oggetto di tipo `Punto` è anche oggetto di tipo `Object`, pertanto entrambe le segnature soddisfano la condizione espressa dalla richiesta di esecuzione.
Questa considerazione ci serve ad asserire che, per evitare ambiguità, è opportuno mantenere solo il metodo più generale e quindi quello che ricopre il metodo `equals` di `Object`. Quanto detto vale per ogni classe, se si dichiara un metodo `equals` è opportuno dichiararlo con argomento di tipo `Object`, cioè ricoprire il metodo `equals` di `Object.`

Il metodo `hashCode`
Fornisce un intero corrispondente (legato) ad un oggetto. La corrispondenza deve soddisfare le condizioni seguenti:

- se due oggetti risultano uguali secondo il metodo `equals`, essi devono fornire lo stesso intero quando viene loro richiesta l'esecuzione di `hashCode;`
- non è richiesto che due oggetti diversi secondo `equals` forniscano interi diversi quando viene loro richiesta l'esecuzione di `hashCode;` è comunque auspicabile che il programmatore progetti il metodo in modo da ottenere interi diversi per oggetti diversi;
- ogni volta che, durante l'esecuzione di una applicazione, l'esecuzione del metodo viene richiesta allo stesso oggetto il metodo deve fornire come risultato lo stesso valore intero. L'intero può essere positivo, negativo o 0. Non è richiesto che venga fornito lo stesso intero in esecuzioni differenti della stessa applicazione;

Il metodo `hashCode` di `Object` fornisce interi distinti per oggetti distinti. Esso è tipicamente implementato convertendo l'indirizzo dell'oggetto in un intero ma questa tecnica implementativa non è richiesta dal linguaggio.
Per soddisfare la prima condizione è necessario che una classe che ricopre `equals`, ricopra anche `hashCode`.
Nella classe `String, hashCode` è ricoperto.

Il metodo `toString`
Ciò che è richiesto a `toString` è che esso fornisca una stringa che rappresenta "testualmente" l'oggetto a cui è richiesto il metodo. Il `toString` di `Object` fornisce una stringa costituita dal nome della classe, seguito da @, seguito dalla rappresentazione esadecimale dell'hashCode dell'oggetto.
E' buona norma ricoprire sempre il metodo `toString`. Nelle classi `String` e `StringBuffer,` esso è ricoperto.

Ereditarietà e Costruttori

Ogni classe eredita dalle superclassi, e le classi che non estendono nessuna classe ereditano da `Object.` Ciò significa che ogni oggetto che viene costruito contiene dentro di sé un esemplare della superclasse diretta che contiene un esemplare della superclasse diretta che contiene… che contiene un esemplare di `Object` e, se eredita solo da `Object`, avrà dentro di sé un oggetto di tipo `Object`. Abbiamo detto che i costruttori non si ereditano, vediamo come avviene la costruzione incrementale di un oggetto.

Il primo statement del corpo di un costruttore di una classe dovrebbe effettuare un'invocazione esplicita ad un costruttore della superclasse diretta, ciò mediante la parola chiave **`super`** seguita dalla lista di argomenti del costruttore invocato, posta tra parentesi rotonde.
La classe `Object` ha un solo costruttore privo di parametri:

```
Object(){}
```

Pertanto, tutte le classi che ereditano solo da `Object,` dovrebbero avere come primo statement del corpo dei loro costruttori lo statement:

```
super();
```

che invoca il costruttore di `Object`. Seguendo questa regola dovremmo riscrivere il costruttore della classe `Punto` nel seguente modo:

```
public Punto(double aX, double aY){
  super();
  x=aX;
  y=aY;
}
```

Se una classe estende `Punto`, i suoi costruttori devono avere come primo statement, l'invocazione dell'unico costruttore di `Punto` con argomenti del tipo corretto (due argomenti di tipo `double`):

```
super(aX,aY);
```

Una classe può avere molti costruttori; se non si vuole vincolare le sottoclassi ad invocarne uno particolare è necessario fornire la classe del costruttore privo di paramentri. Un costruttore privo di parametri ha la seguente forma:

```
ClassName(){super();}
```

Nel corpo di un costruttore lo statement `super();` può essere sottinteso, quindi si può scrivere semplicemente:

```
ClassName(){}
```

poiché è implicito che il costruttore privo di parametri di una classe invochi il costruttore privo di parametri della superclasse diretta.
Se volessimo aggiungere il costruttore privo di parametri alla nostra classe `Punto`, dovremmo aggiungere in essa la dichiarazione:

```
public Punto(){}
```

Questo costruttore inizializza le variabili `x` ed `y` a 0, valore di default del tipo `double`.

Il primo statement di un costruttore può anche essere l'invocazione di un altro costruttore della stessa classe.
Come esempio definiamo una sottoclasse di `Punto` che chiamiamo `Pixel`; tale classe rappresenta un punto colorato. Consideriamo solo le variabili di istanza, ed i costruttori della classe; l'unica variabile è il colore, le coordinate `x` e `y` sono ereditate:

```
public class Pixel extends Punto{
  //variabile privata
  private Color col;

  //costruttori
  public Pixel(double aX, double aY, color aColor){
    super(aX, aY);
    col=aColor;
  }
  public Pixel(double aX, double aY){
    this(aX, aY, Color.White);
  }
…
}
```

Il primo costruttore invoca il costruttore di `Punto` con gli argomenti corretti. Il secondo costruttore invoca il primo costruttore utilizzando la parola chiave `this` seguita da una corretta lista di argomenti (due valori di tipo `double` ed uno di tipo `Color`); esso costruisce un pixel di colore bianco, le cui coordinate sono determinate dagli argomenti di tipo `double`.
Se la classe `Punto` fosse dotata del costruttore privo di parametri, potrebbe essere utile avere anche in `Pixel` un costruttore privo di parametri:

```
public class Pixel extends Punto{
…
  //costruttori
  public Pixel(){
    super();
    col=Color.white;
  }
```

```
…
}
```

Il costruttore costruisce un pixel di coordinate x=0, y=0 e di colore bianco.
Se la classe `Punto` è priva del costruttore senza parametri, non posso invocare `super();`, il compilatore segnala un errore. Sono forzata ad invocare `super(aX, aY),` il costruttore di `Punto` con una lista di parametri adeguata.

Riepilogo

- Ogni classe che deve costruire esemplari deve avere uno o più costruttori.
- Ogni costruttore costruisce un esemplare della classe utilizzando una catena di costruttori delle superclassi a partire da `Object.`
- Il primo statement del corpo di ogni costruttore deve essere o l'invocazione di un costruttore della superclasse diretta o di un costruttore della classe stessa che a sua volta invoca un costruttore della superclasse diretta. Se ciò non accade il compilatore assume che il primo statement nel corpo del costruttore sia `super();` e se nella superclasse diretta manca il costruttore privo di parametri, il compilatore segnala errore. Per evitare di fare errori, è opportuno non lasciare dichiarazioni implicite ed invocare sempre un costruttore della superclasse.
- Ogni classe priva di costruttori dichiarati, viene fornita dal compilatore di un costruttore di default privo di parametri che invoca il costruttore privo di parametri della superclasse, se la superclasse non ha il costruttore privo di parametri il compilatore segnala errore. Se la classe è dichiarata `public` anche il costruttore di default è implicitamente dichiarato `public`.

Da quanto detto è evidente che non ha senso dichiarare un costruttore `final` o `abstract.` Non servirebbe dichiararlo `final` poiché il fatto che i costruttori non possano essere ereditati impedisce che si possano ricoprire. Non ha senso dichiararlo `abstract` poiché un costruttore serve per costruire oggetti e necessita pertanto di un'implementazione. E' inoltre ovvio che un costruttore non può essere dichiarato `static` poiché serve per costruire esemplari di una classe.
Una classe può dichiarare i suoi costruttori `private` ed impedire in tal modo che essi vengano invocati da codice esterno alla classe; può dichiarare metodi pubblici di classe, metodi `static` che invocano il costruttore.
Metodi che creano esemplari di una classe sono detti factory methods.
Per esempio potremmo dichiarare privato il costruttore di `Punto` e dichiarare un metodo `static` che restituisce un oggetto di tipo `Punto`. Es:

```
public static Punto makePunto(double aX, double aY){
  return new Punto(aX,aY);
}
```

Chi ha bisogno di un oggetto di tipo `Punto` dovrà scrivere:

```
Punto p=Punto.makePunto(5,5);
```

invece di:

```
Punto p=new Punto(5,5);
```

Un esempio di ereditarietà

Come esempio di ereditarietà consideriamo una sottoclasse di `Punto` che chiamiamo `Pixel`. La classe realizza il seguente modello:

- un pixel è un punto che ha un colore;
- un pixel è in grado di fornire informazioni sul suo colore;
- un pixel è in grado modificare il suo colore;
- un pixel è in grado verificare se è uguale o diverso da un altro pixel.

La classe `Pixel` deve dichiarare una variabile di tipo `Color` ed i metodi per operare su oggetti di tipo `Color`; inoltre deve ricoprire il metodo `equals` di `Punto`, poiché due pixel sono uguali se hanno le stesse coordinate e lo stesso colore.

```
import java.awt.Color; //necessario per invocare metodi della classe Color

public class Pixel extends Punto{
```

```
  // variabili private
  private Color color;

  // costruttore
  Pixel(double aX, double aY, Color aColor){
    super(aX, aY);
    color=aColor;
  }

  //metodi pubblici
  public int getRed(){
    return color.getRed();
  }
  public int getBlue(){
    return color.getBlue();
  }
  public int getGreen(){
    return color.getGreen();
  }
  public void setColor(int r, int g, int b){
    color=new Color(r,g,b);
  }
  public void setColor(int rgb){
    color=new Color(rgb);
  }
  public boolean equals(Object obj){
    if (!super.equals(obj)) return false;
    else {
      Pixel p = (Pixel)obj;
      return color.equals(p.color);
    }
  }
  public String toString(){
    StringBuffer buf = new StringBuffer();
    buf.append('(');
    buf.append(String.valueOf(getX()));
    buf.append(',');
    buf.append(String.valueOf(getY()));
    buf.append(';');
    buf.append(color.toString());
    buf.append(')');
    return buf.toString();
  }
 }
```

Osserviamo che il metodo `equals` ricopre il metodo `equals` di `Punto` estendendolo; infatti in esso il primo statement invoca il metodo della superclasse diretta.
Il metodo `toString` è equivalente al metodo che segue:

```
  public String toString(){
        return("( "+ getX() + "," + getY() + ";" + color.toString()+")");
}
```

E' opportuno scrivere il corpo del metodo `toString` usando uno `StringBuffer` quando la stringa che si costruisce è troppo lunga per stare su una riga o nel caso in cui si vogliano verificare condizioni.

Considerazioni implementative sull'esecuzione di programmi

Quando si dichiara una variabile viene allocato in memoria lo spazio necessario a rappresentarne il valore.
Se la variabile è di uno dei tipi primitivi è perfettamente noto lo spazio necessario a rappresentare ogni valore del tipo (es: 4 byte per ogni valore di tipo `int`). Quando scriviamo:

```
int x=3;
```

alla variabile x è riservato in memoria uno spazio di 4 byte. Se scriviamo:

```
int x=3;
x=5;
```

all'identificatore x è associato prima il valore 3 poi il valore 5, pertanto nell'area di memoria che rappresentava il valore 3 viene rappresentato il valore 5. Per questa ragione di tipo implementativo, si dice che le variabili di un tipo primitivo "contengono un valore" del tipo. Quando scriviamo:

```
int x;
```

la variabile x non denota alcun valore intero.
E quando si dichiara una variabile di tipo array di o di un tipo class cosa accade? Quando dichiariamo:

```
X x;
```

dove X denota una classe, introduciamo un nuovo termine x e diciamo che tale variabile, nel contesto che stiamo considerando, è riservata per oggetti di tipo X. La dichiarazione fa si che x non denoti alcun oggetto della classe X, ad x è assegnato il valore null.

Gli oggetti possono essere strutture molto complesse è pertanto utile rappresentarli in memoria separatamente dalle variabili che li denotano. Ad una variabile che denota un oggetto è riservato in memoria lo spazio necessario a contenere un numero che rappresenta , nella memoria del calcolatore, l'indirizzodell'oggetto a cui la variabile si riferisce. Si dice che la variabile "punta all'oggetto", essa contiene l'indirizzo dell'oggetto.
Se la variabile non denota alcun oggetto contiene un valore particolare assegnato a null.
Per questa ragione implementativa i tipi dichiarati mediante la dichiarazione di classe sono detti **tipi reference**.

Affinchè x denoti un oggetto della classe, questo deve essere creato ed assegnato a x; lo statement

```
x = new X(…);
```

significa che è stato costruito un oggetto della classe X e che esso viene denotato dalla variabile x.
Quando si crea un nuovo esemplare di una classe in memoria viene allocato lo spazio per tutte le variabili di istanza dichiarate nella classe e di tutte le variabili di istanza dichiarate in ogni superclasse della classe, inoltre le variabili sono inizializzate al loro valore di default. Quindi il costruttore invocato per costruire il nuovo esemplare procede alla costruzione dell'oggetto invocando il costruttore della superclasse e questo il costruttore della superclasse… fino a che si giunge al costruttore di Object ed il processo di invocazione dei costruttori termina. Ogni costruttore, a partire dal costruttore di Object esegue quindi gli statement del suo corpo, quando l'ultimo costruttore ha terminato il suo compito il nuovo oggetto è costruito.
Se immediatamente dopo aver assegnato alla variabile x un oggetto di tipo X facciamo un altro assegnamento:

```
x = new X(…);
```

il primo oggetto generato ed assegnato ad x non ha più alcun termine che si riferisce ad esso, esiste nella macchina ma non se ne può più parlare, non è più accessibile, è spazzatura (**garbage**) e prima o poi deve essere eliminato.
Naturalmente variabili diverse possono denotare lo stesso oggetto. Se per esempio si scrive:

```
X x,y;
x=new X(…);
y=x;
```

le variabili x e y denotano lo stesso oggetto di tipo X. Se agli statement scritti sopra si fa seguire lo statement:

```
x=new X(…);
```

l'oggetto prima denotato da x e da y è ora denotato solo da y, non è pertanto spazzatura.

Quando un oggetto non è più accessibile, è cioè rappresentato in memoria ma non c'è più alcun termine che lo denota, esso viene eliminato mediante un processo automatico detto di **garbage collection** .

Alcuni linguaggi a oggetti, e tra questi il C++, richiedono che sia chi programma a tenere traccia di tutti gli oggetti creati ed a provvedere a distruggerli quando essi non sono più necessari (non c'è alcun riferimento ad essi). Si richiede cioè al programmatore di gestire la memoria della macchina. Poiché la gestione della memoria è estremamente noiosa ed è molto facile commettere errori, Java è stato progettato in modo da evitare la gestione diretta della memoria ad opera del programmatore. Java consente di creare un numero di oggetti limitato solo dalla memoria del sistema su cui opera la Java Virtual Machine (JVM) e di non preoccupparsi della loro eliminazione; gli oggetti sono distrutti automaticamente, a run time, quando essi non sono più accessibili. La **garbage collection**, è fatta tenendo traccia degli oggetti per i quali nel codice da eseguire c'è ancora un riferimento, gli altri sono considerati spazzatura (garbage) ed eliminati.

Se si costruiscono oggetti molto complessi può essere opportuno dichiarare esplicitamente che non servono più e che sono quindi candidati alla eliminazione; a tal fine, se `x` denota un oggetto che si vuole eliminare assegnando a `x` il valore `null` si dice che per l'oggetto denotato da `x` non c'è più alcun riferimento. Con lo statement:

```
x=null;
```

si rende la memoria occupata dall'oggetto potenzialmente disponibile, non c'è però nessun modo di sapere quando il garbage collector la renderà effettivamente disponibile.

Eseguire un programma Java

Descriviamo ad alto livello cosa accade quando viene eseguito un programma Java.
Un programma scritto in un linguaggio object oriented, e quindi anche in Java, è costituito da un insieme di classi e può essere schematicamente così descritto:

- Classe X: se mi chiedono di fare ciò che i miei metodi consentono, lo faccio
- Classe Y: se mi chiedono di fare ciò che i miei metodi consentono, lo faccio
- Classe Z: se mi chiedono di fare ciò che i miei metodi consentono, lo faccio
- …

e nessuna classe fa niente poiché nessuno ne invoca i metodi.

C'è bisogno di qualcuno che invochi un metodo di una delle classi; è necessario fornire un punto di ingresso.
A tal fine viene invocata l'esecuzione di un metodo particolare, il metodo **`main`**, la cui segnatura è la seguente:

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

Il metodo `main` è il metodo che fa partire tutto; esso a sua volta invocherà i metodi necessari alla realizzazione del compito per cui l'applicazione è stata progettata. Pertanto, nell'insieme di classi che costituisce un programma Java, una classe deve contenere il metodo main. Il metodo `main` è `public` perché deve essere invocato dall'esterno della classe, è `static` perché la sua esecuzione non è richiesta a nessun esemplare, non fornisce alcun valore e ciò è indicato con `void`.

Il modo in cui la classe che contiene il metodo `main` è specificata dal sistema operativo alla Java Virtual Machine (JVM) non ci interessa qui.

Interfacce

Cos'è un'interfaccia?
In generale con la parola interfaccia si intende un dispositivo usato da entità tra loro non relate per interagire. Supponiamo di avere di fronte a noi una scatola nera con dei bottoni, l'unico modo che abbiamo di interagire con la scatola per indurla a fare qualche cosa è premere uno dei suoi bottoni. Ad ogni bottone corrisponderà, all'interno della scatola, una o più parti che eseguono la funzione che può essere richiesta premendo il bottone. Per chi interagisce con la scatola, l'informazione relativa alla struttura interna è irrilevante; è rilevante invece conoscere quale funzione corrisponde ad ogni bottone. Ciò che abbiamo esemplificato con la scatola nera può essere detto di ogni oggetto: esso può essere noto ad un altro solo attraverso la sua interfaccia con l'ambiente. Possiamo pertanto asserire che per interagire con un oggetto è necessario conoscere la sua interfaccia. Non c'è nessun modo di interrogare un oggetto o di chiedergli di eseguire un'azione se la sua interfaccia non lo consente. L'interfaccia non dice nulla dell'implementazione dei servizi forniti dall'oggetto, essa definisce un protocollo di comunicazione.

Consideriamo la classe `Punto` e chiediamoci quali sono le azioni che ad un oggetto di tipo `Punto` possono essere richieste. Sono le azioni dichiarate dai metodi pubblici di istanza (non `static`):

```
public double getX()
public double getY()
public void moveTo(double aX, double aY)
public void translateOf(double deltaX, double deltaY)
public boolean equals(Object obj)
public double distanceFrom(Punto p)
```

L'insieme di queste dichiarazioni (le segnature dei metodi pubblici) costituisce l'**interfaccia** della classe `Punto`, l'interfaccia definisce perfettamente il **tipo** `Punto`. Ciò che ci aspettiamo da ogni oggetto di tipo `Punto` è che esso sia in grado di eseguire queste operazioni, non ci interessa sapere come le esegue.

E' evidente che quando si dichiara una classe se ne dichiara implicitamente l'interfaccia, ma è spesso utile poter definire le norme di comportamento degli oggetti, il loro tipo, senza preoccuparsi delle implementazioni. Java consente di fare ciò prevedendo dichiarazioni di tipo che contengono solo dichiarazioni di segnature di metodi pubblici e dichiarazioni di costanti; dei metodi deve essere precisato anche il return type. Ciò è possibile utilizzando la parola chiave **`interface`**. Per esempio possiamo scrivere:

```
public interface Ipunto{
  double getX();
  double getY();
  void moveTo(double aX, double aY);
  void translateOf(double deltaX, double deltaY);
  boolean equals(Object obj);
  double distanceFrom(Punto p);
}
```

L'interfaccia fornisce una dichiarazione astratta di un insieme di servizi, non specifica nulla delle classi i cui esemplari possono fornire quei servizi. I metodi dichiarati sono implicitamente `public`. Essi sono evidentemente `abstract`, il loro corpo è rappresentato dal ";".

Abbiamo detto che la dichiarazione di una classe e la dichiarazione di un "array di" (typeName[]) definiscono un nuovo tipo; in entrambi i casi la definizione porta con sé anche elementi implementativi. Quando si dichiara un interfaccia si definisce un insieme di operazioni, esse possono essere implementate da classi diverse.
Una classe che implementa un'interfaccia deve dichiararlo esplicitamente mediante la parola chiave **`implements`**. Per esempio, una classe che implementa `Ipunto` può essere proprio la classe `Punto`, dobbiamo però modificarne la dichiarazione in questo modo:

```
public class Punto implements Ipunto{…
}
```

Il corpo della classe resta identico a quello che abbiamo già scritto.

Quale può essere il vantaggio di distinguere nettamente la dichiarazione dei metodi pubblici di `Punto` dalla loro implementazione?
Su questo semplice esempio, che ci è servito solo per mostrare la dichiarazione di un'interfaccia, ciò non è evidente. Tuttavia, se si devono fare progetti non banali, la separazione della dichiarazione dei servizi dalla dichiarazione delle classi che li realizzano diventa un elemento essenziale di architettura.
Dichiarata un'interfaccia, si possono dichiarare `n` classi diverse che la implementano, che si impegnano cioè a fornire i servizi specificati dall'interfaccia. Non è sufficiente che una classe che intende fornire i servizi dichiarati da un'interfaccia implementi tutti i metodi dell'interfaccia, essa o una delle sue superclassi, deve dichiarare esplicitamente che implementa l'interfaccia (se `Punto` implementa `Ipunto,` e `Pixel` è una sottoclasse di `Punto,` anche `Pixel` implementa `Ipunto`).
I metodi dichiarati in un'interfaccia sono astratti. Se una classe concreta dichiara di implementare un'interfaccia, deve fornire l'implementazione per ognuno dei metodi dichiarati dall'interfaccia. Può dichiarare `final` i metodi che non vuole vengano ricoperti dalle sue sottoclassi.
Se una classe astratta dichiara di implementare un'interfaccia, essa deve comunque ridichiarare i metodi astratti che non implementa.

Polimorfismo

Poiché ogni interfaccia definisce un tipo, si può dichiarare una variabile il cui tipo è definito da un'interfaccia.
Che valori può denotare una tale variabile? Essa denoterà un oggetto, ma quale oggetto? Qualsiasi oggetto che sia esemplare di una classe che dichiara di implementare l'interfaccia (un oggetto è sempre esemplare di una classe).

Da quanto detto è evidente che se classi diverse implementano la stessa interfaccia, i loro esemplari condividono un tipo, il tipo definito dall'interfaccia. Inoltre, ognuno dei loro esemplari ha almeno due tipi: quello della propria classe e quello dell'interfaccia che la classe implementa. Può naturalmente avere altri tipi, tutti quelli delle superclassi da cui la classe di cui è esemplare eredita (quindi almeno il tipo `Object`).

Poiché un oggetto può avere molti tipi, ad una variabile di un tipo reference (`interface, class, []`) possono essere assegnati oggetti di classi diverse a patto che essi siano anche del tipo della variabile (una variabile di tipo `EssereVivente` può denotare un esemplare di tipo `Leone` ma anche un esemplare di tipo `Quercia,` un leone ed una quercia sono anche di tipo `EssereVivente`). Ciò ha come conseguenza che l'esecuzione di un metodo a run time dipende dall'oggetto a cui tale esecuzione è richiesta. L'oggetto cercherà nella sua classe e se in essa il metodo non è definito cercherà nella superclasse diretta e poi nella superclasse della superclasse diretta fino a quando non trova un'implementazione. La stessa operazione (metodo, servizio) ha molte forme.

Ricordiamo che gli array si costruiscono con elementi di qualsivoglia tipo (il tipo dell'array è `TypeName[]`), si possono pertanto dichiarare array i cui elementi siano del tipo definito da un'interfaccia. In tal caso gli elementi dell'array avranno come valori esemplari di classi che dichiarano di implementare l'interfaccia oppure ad essi sarà assegnato `null`. E' evidente che si possono anche dichiarare array i cui elementi siano del tipo definito da una classe astratta; gli elementi dell'array avranno come valori esemplari di sottoclassi concrete della classe astratta oppure ad essi sarà assegnato `null`.

Esempi di interfacce

Consideriamo alcuni esempi di interfaccia. Supponiamo di definire il tipo `Soggetto` mediante un interfaccia. Diciamo che un soggetto ha un nome, un indirizzo e un codice fiscale; l'interfaccia dichiarerà pertanto tre metodi:

```
public interface Soggetto {
  String getName();
  String codiceFiscale();
  String indirizzo();
}
```

Abbiamo precisato cosa possiamo chiedere ad un esemplare di tipo `Soggetto.` Chi può essere un esemplare di tipo `Soggetto`? Possiamo dire che una persona è un soggetto e che un'impresa è un soggetto; in tal caso abbiamo almeno due classi che implementano l'interfaccia, le chiamiamo `Persona` e `Impresa`. Ognuna di esse implementerà in modo diverso almeno il metodo `codiceFiscale` poiché l'algoritmo per calcolarlo è diverso per persone e imprese. Si può pertanto scrivere:

```
Soggetto s1,s2;
String c1,c2;
s1= new Persona();
s2=new Impresa();
c1=s1.codiceFiscale();
```

```
c2=s2.codiceFiscale();
```

Supponiamo di dichiarare classi che realizzano modelli di diverse figure geometriche (triangoli, rettangoli, poligoni,…) e di voler asserire che alcune di queste sono in grado di visualizzarsi sullo schermo ed altre no. Potremmo definire un'interfaccia che viene implementata solo dalle classi che consentono ai loro esemplari di disegnarsi:

```
public interface Drawable {
  void draw();
}
```

Se dichiarassimo che la classe `Rettangolo` implementana l'interfaccia `Drawable`, potremmo chiedere ad una variabile dichiarata di tipo `Rettangolo` di eseguire il metodo `draw`. Es:

```
public class Rettangolo implements Drawable {
  …
}
```

La classe implementando l'interfaccia si dichiara in grado di disegnarsi, pertanto possiamo chiedere ad un rettangolo di disegnarsi:

```
Rettangolo r = new Rettangolo();
r.draw();
```

L'interfaccia `comparable`

In Java è definita un'interfaccia, `Comparable` che impone a tutte le classi che la implementano di definire un ordinamento totale sui propri esemplari. `Comparable` è in generale implementata da classi i cui oggetti hanno un ordinamento naturale. Liste e array di oggetti le cui classi implementano `Comparable` possono essere ordinati automaticamente. L'interfaccia consiste di un solo metodo:

```
public interface Comparable {
  int compareTo(Object obj);
}
```

Ogni classe che dichiara di implementare `Comparable` è libera di definire i criteri secondo cui ordinare i suoi elementi (es: una classe `Persona` potrebbe scegliere, l'altezza, l'età…); fissato un ordinamento, essa deve però garantire che l'implementazione di `compareTo` soddisfi le seguenti richieste :

- se questo oggetto precede `obj` (l'argomento del metodo) viene fornito un intero negativo; se questo oggetto segue `obj` viene fornito un intero positivo;
- se questo oggetto è uguale a `obj` viene fornito il valore 0;
- se si inverte l'ordine di comparazione, il segno dei risultati sarà opposto.

Altrimenti detto, se `x, y` e `z` sono esemplari di una classe che dichiara di implementare `Comparable` e **signum**(x) è la funzione che fornisce –1 se x<0, 1 se x>0, 0 se x= 0 deve valere che:

- **signum**`(x.compareTo(y))= -`**signum**`(y.compareTo(x))`
- se vale `(x.compareTo(y)>0 & y.compareTo(z)>0)`, allora `x.compareTo(z)` deve fornire un intero maggiore di `0.`
- se vale `(x.compareTo(y) == 0)`, e `x.compareTo(z)` è maggiore (minore) di 0, allora `y.compareTo(z)` deve essere maggiore (minore) di 0. In altre parole: **signum**`(x.compareTo(z))=` **signum**`(y.compareTo(z))`

Inoltre è fortemente raccomandato che il metodo `compareTo` dichiari due oggetti uguali se e solo se lo sono rispetto al metodo `equals`. Più precisamente deve valere:

```
(x.compareTo(y)==0)==x.equals(y)
```

Un ordinamento che gode di questa proprietà è detto essere **consistente con** `equals`.
Classi di Java che implementano `Comparable` sono: `String, Byte, Character, Short, Integer, Long, Float, Double, File, Date`.

L'interfaccia `cloneable` e il metodo `clone` di `Object`

In `Object` è definito il metodo `clone` la cui segnatura è:

```
protected Object Clone()
```

Mediante il metodo `clone` un oggetto fornisce una copia di se stesso; cosa vuol dire "copia di se stesso"?
Riflettiamo sul significato dell'operazione "copia".
Copiare un oggetto significa costruire un un nuovo esemplare della stessa classe con la stessa struttura interna dell'oggetto copiato. Cosa vuol dire con la stessa struttura interna? Distinguiamo due diverse modalità di copiare la struttura interna di un oggetto e le indichiamo rispettivamente con *copia superficiale* (shallow copy) e con *copia profonda* (deep copy).
Effettuare una *copia superficiale* di un oggetto `x` della classe `C` significa creare un nuovo esemplare di `C` e inizializzarne tutte le variabili di istanza con gli stessi valori delle variabili di `x` (l'oggetto copiato) come in una operazione di assegnamento. Altrimenti detto, supponiamo di dichiarare un metodo `clone` nel seguente modo:

```
public Object Clone(){
  C xCopy= new C();

  //per ogni variabile di istanza v
  xCopy.v = x.v;
  return xCopy;
}
```

Se `x` denota un automobile con 4 ruote, `xCopy` denota una nuova automobile con le stesse 4 ruote. Dopo avere eseguito la copia di `x` abbiamo pertanto un totale di due automobili e un totale di 4 ruote, la seconda automobile, `xCopy,` ha le stesse ruote della prima. In generale non si vuole ottenere questo risultato; sarebbe opportuno che la nuova automobile avesse proprie ruote, è pertanto necessario che venga creato un nuovo esemplare anche delle ruote. Non basta copiare l'oggetto in modo superficiale.
Immaginiamo un metodo, `deepCopy`, in grado di eseguire una copia profonda di un oggetto:

```
public Object deepCopy(){
  C xCopy= new C();

  //se v è di tipo primitivo
  xCopy.v=x.v;

  //se v è di tipo reference
  Object y=x.v;
  if(y==null) xCopy.v=null;
  else xCopy.v= y.deepCopy();
  return xCopy;
}
```

Le variabili dei tipi primitivi vengono inizializzate con gli stessi valori delle variabili di `x` (l'oggetto copiato), le variabili dei tipi reference possono essere a loro volta copiate, ciò se la classe a cui tali oggetti appartengono ha un metodo `deepCopy` che ne consente la copia. Ciò che abbiamo scritto sembra sensato, ma supponiamo che esista una variabile di istanza `madeIn` di tipo `Country`. Ogni volta che si copia un oggetto costruito in Italia si copia l'Italia.
E' evidente che ogni variabile deve essere trattata in modo opportuno, pertanto effettuare una *copia profonda* di un oggetto `x` della classe `C` significa creare un nuovo esemplare di `C` e trattare in modo diverso le variabili dei tipi primitivi dalle variabili dei tipi reference, di queste è necessario chiedersi quali devono essere a loro volta copiate e quali no.
Per ogni classe che vuole consentire di effettuare copie dei suoi oggetti e' necessario chiedersi di quale metodo di copiatura si deve dotare, e per le variabili di tipo reference quali devono essere a loro volta copiate e perché.

Il metodo `clone` implementato in `Object` realizza una shallow copy di un oggetto. L'intento generale di `clone` è che per ogni oggetto siano vere le espressioni:

```
x.clone()!=x
x.clone().equals(x)
```

Poiché ogni classe eredita da `Object` si potrebbe pensare che ogni classe può clonare i suoi esemplari. Non è così. Affinchè agli oggetti di una classe possa essere richiesta l'esecuzione di `clone,` la classe deve dichiarare di implementare l'interfaccia `Cloneable:`

```
public interface Cloneable{}
```

che non dichiara alcun metodo. E' una marker interface, ogni classe che la implementa dichiara semplicemente di poter clonare i suoi esemplari; per farlo può invocare il metodo `clone` di `Object` o ridichiarare `clone` ricoprendolo in modo opportuno.
Se una classe non dichiara di implementare `Cloneable,` quando ad un suo esemplare viene richiesto di eseguire `clone` esso lancia un messaggio di errore. Il metodo `clone` può essere richiesto solo ad oggetti di tipo `Clonable`, cioè ad esemplari di classi che dichiarano di implementare l'interfaccia `Cloneable.` La classe `Object` non implementa `Cloneable` quindi non si può richiedere ad un oggetto di tipo `Object` di clonarsi.
Il metodo `clone` di `Object` e' stato costruito solo come metodo di utilità per le sottoclassi che intendono effettuare copie superficiali dei loro esemplari; la implementazione di default evita a chi definisce nuove classi di scrivere un metodo per la copiatura di esemplari se l'azione di `clone` è adeguata per la classe.

Ereditarietà di interfacce

Un'interfaccia può essere dichiarata l'estensione diretta di una o più interfacce; non ci sono vincoli sul numero di interfacce che un'interfaccia può estendere direttamente:

```
public interface A extends B,C,D{…}
```

Se un'interfaccia estende direttamente altre interfacce essa implicitamente dichiara tutto ciò che è dichiarato nelle interfacce che estende, ne eredita tutti i metodi e tutte le costanti. Nel nostro esempio `A` eredita i metodi e le costanti dichiarati in `B,C` e `D,` le superinterfacce dirette (direct superinterfaces) di `A.`

Un'interfaccia può dichiarare anche solo costanti; facciamo un semplice esempio di ereditarietà di interfacce usando interfacce che dichiarano solo costanti (è convenzione scrivere i nomi delle costanti con caratteri maiuscoli):

```
public interface BaseColorsConstants{
  int[]RED={255,0,0}, GREEN={0,255,0} BLUE={0,0,255};
}

public interface PrintColorsConstants extends BaseColorsConstants{
  int[]YELLOW={255,255,0}, CYAN={0,255,255} MAGENTA={255,0,255}
}

public interface RainbowColorsConstants extends BaseColorsConstants{
  int[]ORANGE={255,100,0}, INDACO={0,100,255} VIOLET={100,0,255}
}

public interface LotsOfColorsConstants extends PrintColorsConstants,
RainbowColorsConstants {
  int[]FUCHSIA={255,0,100}, BLACK={0,0,0}, WHITE={255,255,255}
}
```

Se una classe dichiara di implementare `LotsOfColorsConstants` essa ha a disposizione, senza ridichiararle, tutte le costanti dichiarate nelle interfacce sopra definite; in essa si può scrivere:

```
public class Aclass implements LotsOfColorsConstants {
…
 int[] myColor = new int[3];
 for(int i=0; i<myColor.length; i++) {
  myColor[i] = RED[i] + INDACO[i];
  System.out. println("la "+i+"componente del mio colore è: "+myColor[i]);
  }
…
}
```

Qualsivoglia classe `C,` che implementa un'interfaccia `I,` di fatto implementa anche tutte le interfacce che `I` estende.

L'interfaccia `LotsOfColorsConstants` specifica, oltre ai colori dichiarati, tutti i colori delle interfacce che estende direttamente e queste a loro volta quelli dell'interfaccia che estendono direttamente.
Se un'interfaccia estende direttamente diverse interfacce, può accadere che essa erediti più volte la stessa costante; nell'esempio sopra, le costanti `red, green, blue` sono ereditate da `LotsOfColorsConstants` sia attraverso

`PrintColorsConstants` che attraverso `RainbowColorsConstants.` Ciò non pone problemi, non c'è alcuna ambiguità poiché alle costanti `red, green, blue` è assegnato lo stesso valore nelle due interfacce (entrambe estendono `BaseColorsConstants`).
Può tuttavia accadere che un'iterfaccia `I`, che estende direttamente due interfacce `A` e `B`, erediti da esse due costanti con lo stesso nome e con due valori diversi (es: due diverse approssimazioni Π); in tal caso non ci si può riferire a `I.`Π poiché sarebbe ambiguo, è sempre necessario riferirsi a: `A.`Π oppure `B.`Π`.` Occorre evitare dichiarazioni che possono produrre ambiguità.

Una interfaccia che estende altre interfacce può ridichiare le costanti (hiding di costanti) delle superinterfacce, e dichiarare metodi con lo stesso nome e diversa segnatura dei metodi delle superinterfacce (overloading di nomi). L'overloading di nomi non provoca problemi poiché i metodi con lo stesso nome hanno diverse segnature e non è richiesta nessuna relazione tra i loro return type. La ridichiarazione di costanti è da evitare.

Per chiusura transitiva della relazione è superinterfaccia diretta di si ottiene la relazione è superinterfaccia di.
Un'interfaccia `K` è superinterfaccia di un'interfaccia `J` se vale una delle seguenti condizioni:

- `K` è superinterfaccia diretta di `J`
- Esiste un'interfaccia `I` tale che `K` è superinterfaccia di `I,` e `I` è superinterfaccia di `J,` applicando la definizione ricorsivamente

Un'interfaccia `J` è detta sottointerfaccia (subinterface) di `K` se `K` è superinterfaccia di `J`.

Osservazione: ogni classe estende `Object` (ogni esemplare è un oggetto che deve essere costruito) che è la radice della gerarchia di classi; non c'è nessuna interfaccia di cui tutte le altre sono un'estensione.

Una classe può implementare molte interfacce

Le interfacce servono a dichiarare esplicitamente i servizi che un tipo può fornire. Una classe ha una sola superclasse diretta ma può implementare più interfacce.
Immaginiamo di progettare una classe `Rettangolo` i cui esemplari siano in grado di disegnarsi, compararsi con altri rettangoli (supponiamo di definire una relazione di ordinamento tra rettangoli), e costruire una copia di se stessi; la classe potrebbe essere così dichiarata:

```
public class Rettangolo implements, Drawable, Clonable, Comparable {
…
}
```

`Rettangolo` deve implementare i metodi `draw`, `compareTo` e, molto probabilmente, ricoprire `clone`.
Un oggetto della classe `Rettangolo` è di tipo `Rettangolo,` di ognuno dei tipi delle superclassi di `Rettangolo` (quindi sicuramente di tipo `Object),` di tipo `Drawable,` di tipo `Clonable,` di tipo `Comparable.`

Riepilogo

- La dichiarazione di un' interfaccia definisce un tipo. Il corpo di un'interfaccia è costituito dalla dichiarazione di costanti e di metodi astratti.
- Un'interfaccia deve essere implementata; in generale è implementata da più classi. I valori del tipo definito dall'interfaccia sono gli oggetti delle classi che implementano l'interfaccia. Se una classe fornisce l'implementazione di tutti i metodi di un'interfaccia, ma non dichiara esplicitamente di implementarla, non è considerata fornire un'implementazione dell'interfaccia.
- Una classe può dichiarare di implementare direttamente una o più interfacce. Se una classe `C` implementa direttamente più interfacce, supponiamo `A` e `B`, i suoi esemplari sono di tipo `C`, ma anche di tipo `A` e di tipo `B`.
- Un'interfaccia può estendere direttamente una o più interfacce; ciò significa che essa specifica implicitamente tutti i metodi e le costanti delle interfacce che estende. Un'interfaccia `A,` che estende direttamente un'interfaccia `B,` può dichiarare metodi con lo stesso nome ma con segnature diverse (overloading di nomi), può ridichiarare le costanti (hiding di costanti).
- Una classe `C` che implementa un'interfaccia `T` deve necessariamente implementare tutte le interfacce che l'interfaccia `T` estende direttamente. Inoltre se `C` estende una classe, deve implementare tutte le interfacce che la sua superclasse diretta implementa
- Se `T` è un' interfaccia, una variabile dichiarata di un tipo `T` può denotare qualsiasi oggetto di una classe che dichiara di implementare l'interfaccia `T`.

Tipi, Interfacce e classi

Le interfacce sono fondamentali in un sistema organizzato ad oggetti. Gli oggetti sono noti solo attraverso le loro interfacce; infatti ad ogni oggetto si può richiedere solo l'esecuzione dei metodi la cui segnatura è dichiarata dalla sua interfaccia. L'interfaccia caratterizza completamente l'insieme di richieste che possono essere fatte all'oggetto. Un oggetto ha molti tipi, e diversi oggetti possono condividere lo stesso tipo. Due oggetti dello stesso tipo devono condividere una parte della loro interfaccia.

Ogni oggetto è creato come esemplare di una classe; è la classe che specifica le variabili, i costruttori e l'implementazione dei metodi che un oggetto può eseguire. L'interfaccia non dice nulla dell'implementazione; classi diverse sono libere di implementare la stessa interfaccia in modi diversi.

E' importante comprendere la differenza che esiste tra la *classe di un oggetto* e il *tipo di un oggetto.* La classe definisce lo stato e l'implementazione dei metodi, per contro il tipo è l'interfaccia: l'insieme di richieste che si possono fare ad un oggetto. Naturalmente una classe definisce anche un tipo; dire che un oggetto è un esemplare di una classe equivale implicitamente a dire che la sua interfaccia è definita dalla classe.
L'ereditarietà tra classi consente di definire l' implementazione di oggetti mediante l'implementazione di altri oggetti (è un meccanismo per condividere codice) l'ereditarietà di interfacce (subtyping) descrive quando un oggetto può essere usato al posto di un altro.

Quando si definisce una classe astratta si definisce anche un'interfaccia comune alle sottoclassi, perché definire la classe e non l'interfaccia? In questo modo si può fornire l'implementazione di alcuni importanti metodi comuni lasciando gli altri da implementare alle diverse sottoclassi concrete.

Packages

Le classi e le interfacce sono organizzate in contesti denominati **packages**.
Un package è un'entità Java che deve essere dichiarata e può avere le seguenti componenti:

- tipi reference (classi e interfacce)
- subpackages (insieme di tipi)

Consideriamo il package il cui nome è `java.lang,` esso contiene classi e interfacce che sono parte integrante del linguaggio Java; `java.lang` contiene:

- `Object`, la radice di tutta la gerarchia di classi Java;
- `String` e `StringBuffer,` classi essenziali per trattare le stringhe;
- `Math`, classe che fornisce metodi statici per il calcolo di funzioni matematiche di uso comune;
- le classi `Boolean, Character, Integer, Long, Float, Double`, che consentono di trattare valori dei tipi primitivi come fossero oggetti;
- classi che consentono di trattare errori ed eccezioni (`Throwable` e le sue sottoclassi);
- classi che forniscono operazioni di sistema quali sono quelle necessarie a trattare le classi caricate dinamicamente, a creare processi esterni, implementare politiche di sicurezza…(`ClassLoader, Process, Runtime, SecurityManager, System`);
- le interfacce `Clonable, Comparable` e `Runnable`.

Ci sono altri packages il cui nome inizia con l'identificatore `java` (`java.lang, java.util, java.io, java.net, java.awt,…),` essi forniscono numerose classi ed interfacce utili allo sviluppo di applicazioni.
Per meglio comprendere la necessità di introdurre i packages è utile fare alcune considerazioni sull'uso dei nomi.

Teoria dei Nomi

Un **nome** è un **termine**, e in quanto tale denota un oggetto (una cosa del mondo reale).
Un nome è generalmente una **costante**, cioè un termine associato ad un oggetto (cosa) in maniera **permanente**.
Per comprendere una frase contenente un nome è necessario associare al nome l'oggetto giusto, proprio quello che nell'intenzione di chi ha emesso la frase è denotato dal nome, e non un altro. Questa è la **risoluzione** di un nome.

Letterali

I letterali sono nomi particolarmente semplici da risolvere: c'è una regola ben definita che associa il nome all'oggetto: anzi consente in generale di "ricostruire" l'oggetto a partire dal nome.

Nomi Propri

I nomi propri hanno sempre generato problemi, perchè non sempre chi interpreta una frase conosce/distingue l'oggetto denotato dal nome.

- Leonardo ...
- Leonardo chi?
- Leonardo da Vinci

Si rendono i nomi più facili da risolvere associando ad essi altri nomi, che denotano un contesto in cui la risoluzione del nome è più facile, e possibilmente certa. Questi nomi associati ad altri nomi sono i **nomi qualificati**:

- Leonardo da Vinci
- Maria di Nazaret
- Alessandro Manzoni
- Paolo Rossi dell'ufficio Acquisti

Nomi qualificati

Scriviamo i nomi qualificati in un modo un po' meno informale:

- Vinci/Leonardo
- Nazaret/Maria
- Manzoni/Alessandro
- UfficioAcquisti/Rossi/Paolo

Un nome qualificato è una lista in cui ciascun elemento (tranne l'ultimo) denota un contesto in cui risolvere l'elemento successivo; l'ultimo elemento individua un particolare oggetto all'interno dell'ultimo contesto individuato.
In questo modo l'operazione di risoluzione si decompone in più sotto-operazioni di risoluzione più semplici.

Risoluzione(E1/E2/.../En) ::=
Risoluzione(E1); // in un contesto universale
Risoluzione(E2); // nel contesto definito al passo precedente
...
Risoluzione(En); // nel contesto definito al passo precedente

Nell'operazione di risoluzione il passaggio più critico è il primo, che avviene in un contesto universale, o contesto-radice.

Battesimo

Maria di Nazaret si chiama Maria perchè qualcuno (i suoi genitori) le hanno assegnato il nome di Maria, e la comunità universale accetta che siano i genitori a determinare il nome dei figli; e la città di Nazaret si chiama Nazaret perchè così avviene da tempi immemorabili e le autorità amministrative della Palestina accettano questa tradizione. Così non avviene sempre: una remota località di pescatori diventa per decisione dello zar la città di Pietroburgo, poi per decisione del governo sovietico Leningrado, poi ancora Pietroburgo per decisione del governo post-sovietico.
Per la comprensione dei discorsi sono utili anche dei sinonimi, anche costruiti con differenti sistemi di contesti:

ImperoRusso/Pietroburgo = URSS/Leningrado = Russia/Pietroburgo

Indirizzi di rete (HOST)

Tutti i calcolatori accessibili in rete hanno un nome qualificato:

- java.sun.com
- repubblica.it
- dsi.unimi.it

L'ordine degli elementi è in questa particolare convenzione rovesciato. Con la sintassi introdotta prima si scriverebbe

- com/sun/java
- it/repubblica
- it/unimi/dsi

I contesti da riconoscere nell'ambito del contesto universale sono **com** e **it.** Contesti come questi vengono battezzati da qualche autorità universalmente riconosciuta. All'interno di *com* si risolve *sun*; all'interno di *it* si risolvono *repubblica* e *unimi*; ...

Indirizzi di rete (risorse): URL (uniform resource location)

Un url è un nome qualificato che identifica una risorsa accessibile in rete (un file, un documento, un'immagine...
Ha la forma convenzionale

//host/contesto1/contesto2/.../contestoN/NomeRisorsa

dove *host* è un indirizzo host di rete (che, come visto sopra, è anch'esso un nome qualificato).

Nomi di file (FileSystem locale)

Si tratta di nomi qualificati dove il primo contesto è un'unità disco, e i contesti successivi sono directory. L'ultimo elemento individua il file.
C:\aaa\bbb\untesto.doc

Nomi in Java

In Java esistono tanti nomi:

- le variabili sono nomi non permanenti
- i nomi dei metodi sono nomi permanenti
- i nomi delle classi sono nomi permamenti
- i nomi dei tipi primitivi sono nomi permanenti
- i nomi dei tipi non primitivi sono anch'essi nomi permanenti.

Alcuni di questi nomi denotano dei contesti:

- i nomi dei metodi
- i nomi delle classi

Fino a che in un certo contesto si usano nomi definiti all'interno dello stesso contesto, non c'è problema.
Ma che cosa succede se nell'ambito della classe MiaClasse voglio nominare una certa classe TheirClass definita da un certo laboratorio della Bell?
Devo in qualche modo definire il contesto in cui questa classe è chiaramente individuabile, e un percorso per tale contesto.
Se ricordiamo il discorso degli URL, con la logica degli URL posso nominare un qualsiasi file in qualsiasi host; quindi anche in quello della Bell.

//bell.maincomputer.com/research/year2000/java/programs/lab34/project56/TheirClass.java

Questo è il nome del file che contiene il codice java. Ma qual è il nome della classe all'interno del linguaggio Java? Potrei ipotizzare che il nome della classe sia qualcosa come l'URL del file senza il suffisso .java (infatti in java le classi non hanno suffisso)

//bell.maincomputer.com*/research/year2000/java/programs*/lab34/project56/TheirClass

Ci siamo quasi, ma non ancora del tutto.

La parte in grassetto è l'indirizzo di un particolare host della Bell. E' importante sapere che si tratta di un programma della Bell, ma non è importante (all'interno del programma) sapere che si tratta del particolare host **maincomputer**.

La parte in corsivo ci guida all'interno del labirinto delle directory del maincomputer. E' essenziale per arrivarci fisicamente oggi, ma potrebbe essere cambiata domani. E non c'entra niente con il mondo Java.

La parte non corsiva sembra più significativa, ma si tratta ancora di un indirizzo fisico.

L'ideale sarebbe che i responsabili della programmazione Java della Bell stabilissero un piano del tipo:

- stabiliamo un contesto per ogni laboratorio, e ciascun laboratorio amministra il proprio contesto.
- all'interno di ciascun laboratorio, si stabilisce un contesto per ogni progetto
- all'interno del contesto di un progetto possono essere aperti sotto-contesti e/o collocati programmi java.

Questi contesti in Java si chiamano **package**.

C'è un contesto della Bell

- com.bell

Nota: c'è scritto "bell" e non "Bell" perchè è stato stabilito per convenzione che i nomi dei package abbiano l'iniziale minuscola.

C'è un contesto per ogni laboratorio; per il laboratorio dei nostri amici si tratta di

- com.bell.lab34

All'interno del loro contesto, i nostri amici hanno aperto il contesto (package)

- com.bell.lab34.project56

All'interno di questo contesto (package) i nostri amici hanno collocato la classe **TheirClass**, che quindi ha un nome universale

- **com.bell.lab34.project56.TheirClass** (***)

Se mi serve scaricare il file con un browser, utilizzerò l'URL riportato sopra; ma per identificare senza ambiguità la classe all'interno del linguaggio basta (e ci vuole) il nome qualificato (***)

Quindi potrò scrivere:

```
...
com.bell.lab34.project56.TheirClass x = new com.bell.lab34.project56.TheirClass();
...
```

C'è ancora un difetto: è troppo lungo.

Soluzione:

in testa al mio programma dichiaro

import com.bell.lab34.project56.TheirClass;

oppure, per intendere "tutte le classi del package"

import com.bell.lab34.project56.*;

e poi:

```
...
TheirClass x = new TheirClass();
...
```

Adesso ci siamo.

Posso nominare le classi scritte da tutti i programmatori del mondo, e utilizzarle senza alcuna ambiguità.

Devo tuttavia ricambiare il favore.

Devo strutturare lo spazio dei miei programmi in modo da non perdermi, e da consentire ai miei amici della Bell di arrivarci.

Scriverò in testa al mio programma la sua collocazione logica:

package edu.unimi.math.corso2000;

Qualunque programmatore del mondo, per usare la mia classe, dovrà scrivere:

import edu.unimi.math.corso2000.*;

Sintassi

Quando si dichiarano classi e interfacce, la dichiarazione del package a cui esse appartengono deve precedere la dichiarazione del tipo che si sta dichiarando. In generale, si pongono nello stesso package classi ed interfacce che servono per realizzare un certo progetto. La dichiarazione di un package ha la forma seguente:

```
PackageDeclaration:= “package” PackageName “;”
```

Per porre nello stesso package, che chiamiamo `esempio`, le classi `Punto`, `Pixel` dobbiamo scrivere:

```
package esempio;
public class Punto {…}

package esempio;
public class Pixel extends Punto {…}
```

Se ci riferiamo al package `esempio`, siamo autorizzati ad utilizzare in ognuna delle sue classi il nome semplice di ogni altra classe dello stesso package. E se dobbiamo riferirci ad un tipo dichiarato in un altro package? Per esempio in `Pixel` abbiamo invocato metodi e costruttori delle classi `String, StringBuffer` e `Color`. Tutte le classi di `java.lang,` sono disponibili e ci si può riferire ad esse con il loro nome semplice; la classe `Color` è posta in un package il cui nome è `java.awt.` Il nome qualificato della classe è pertanto `java.awt.Color`; al fine di riferirsi ad essa con il nome semplice è necessario effettuare una esplicita dichiarazione di importazione (dichiarazione di import) che ha la forma seguente:

```
import java.awt.Color;
```

Se non si fa la dichiarazione di `import` non si può utilizzare il nome semplice `Color`, occorre utilizzare il suo nome qualificato `java.awt.Color`. Se di un package si devono utilizzare molte classi, al fine di evitare un lungo elenco di import statement si importa tutto il package. La sintassi della dichiarazione di import è la seguente:

```
ImportDeclaration:= SingleTypeImportDeclaration|
TypeImportOnDemandDeclaration
SingleTypeImportDeclaration := “import” TypeName “;”
TypeImportOnDemandDeclaration:=“import” PackageName “.” “*” “;”
```

Poiché il package `java.lang` contiene classi fondamentali per ogni programma java, lo statement

```
import java.lang.* ;
```

viene omesso, è implicito.

Riepilogo
Ogni dichiarazione introduce una nuova entità; una dichiarazione include sempre un identificatore.
Le entità che vengono dichiarate in un programma Java sono:

- i packages
- i tipi importati mediante una `ImportDeclaration`
- i tipi reference (interfacce, classi)
- le componenti (variabili e metodi) di classi e interfacce
- le variabili locali
- i parametri formali di metodi e costruttori.

nella dichiarazione dei costruttori non si introduce un nuovo identificatore, si usa l’identificatore della classe.

Ad ogni entità dichiarata ci si riferisce mediante un nome; in Java i nomi hanno due forme: ci sono i **nomi semplici**, che consistono di un singolo identificatore e i **nomi qualificati** che consistono di una sequenza di identificatori separati dal punto “.”
Ogni dichiarazione ha uno **scope**, un ambito di validità, che consiste della parte di codice in cui ci si può riferire all’entità dichiarata mediante il nome semplice.
Lo scope di una `ImportDeclaration` è l’insieme di tutte le classi e interfacce in cui essa appare.
Lo scope di un tipo, introdotto da una dichiarazione di classe o da una dichiarazione di interfaccia, è la dichiarazione di tutte le classi ed interfacce dello stesso package in cui il nuovo tipo è dichiarato.
Lo scope di un componente, dichiarato o ereditato da una classe o un’iterfaccia, è l’intera dichiarazione di classe o interfaccia.
Lo scope di una variabile locale, è il resto del blocco in cui la dichiarazione di variabile appare.
Lo scope di un parametro formale di un metodo è l’intero corpo del metodo.
Lo scope di un parametro formale di un costruttore è l’intero corpo del costruttore.

Convenzioni per i nomi

Ci sono alcune convenzioni per dare i nomi in Java che è consigliato e conveniente utilizzare.

I nomi di packages che devono essere resi largamente disponibili devono sempre essere nomi qualificati, essi dovrebbero essere formati in modo da evitare la possibilità che due packages abbiano lo stesso nome.
I nomi di packages che si intendono utilizzare solo localmente dovrebbero avere un'identificatore che inizia con una lettera minuscola, tale identificatore non può essere `java;` packages che iniziano con `java` sono riservati a chi ha disegnato e sviluppato il linguaggio.
I nomi di classi ed interfacce dovrebbero essere nomi o frasi nominali descrittive, non troppo lunghe, che iniziano con una lettera maiuscola e che contengono lettere maiuscole e minuscole. Es:

```
Number
ClassLoader
BufferedInputStream
Cloneable
DataInput
```

I nomi di metodi dovrebbero essere verbi o frasi verbali che iniziano con una lettera minuscola e che contengono lettere maiuscole e minuscole. Ci sono altre specifiche addizionali:

- metodi che chiedono o fissano il valore di una variabile `v` dovrebbero utilizzare i verbi `get` e `set`, esempio `getV` e `setV`
- un metodo che fornisce la lunghezza di qualcosa dovrebe chiamarsi `length` come nella classe `java.lang.String`;
- un metodo che verifica una condizione booleana `V` circa un oggetto, dovrebbe chiamarsi `isV`;
- un metodo che converte un oggetto ad un particolare formato `F` dovrebbe chiamarsi `toF` come il metodo `toString` della classe `java.lang.Object`;

I nomi di variabili di istanza o di classe dovrebbero iniziare con una lettera minuscola e contenere lettere maiuscole e minuscole. Gli identificatori di variabili possono essere nomi, frasi nominali o abbreviazioni. Esempi sono `buf, count, pos.` In classi ben disegnate, tutte o gran parte delle variabili dovrebbero essere private.

I nomi di costanti (variabili `final`) dovrebbero essere costituiti da una o più parole, acronimi o abbreviazioni, scritte in caratteri maiuscoli eventualmente separati da un underscore. Es: `MAX_VALUE, MIN_VALUE`.

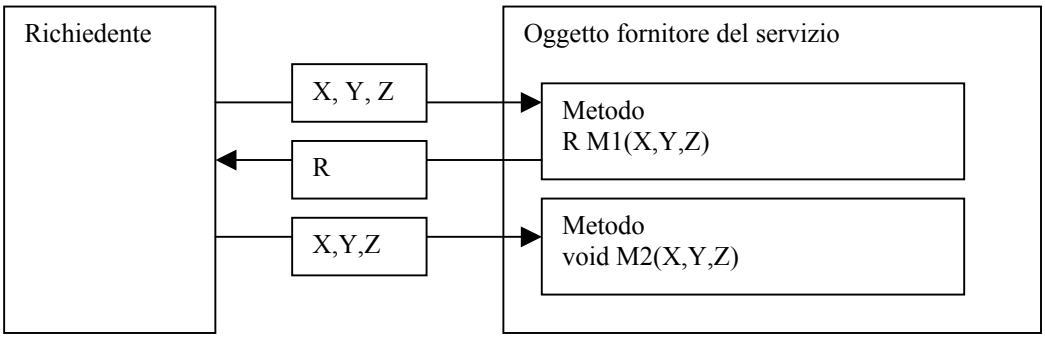
I nomi di variabili locali e di parametri formali dovrebbero essere corti e nonostante questo avere un significato. Variabili locali costituite da un solo carattere dovrebbero essere evitate eccetto che per variabili in cicli `for` o per variabili di tipi particolari, es:

`b` per un `byte`;
`c` per un `char`;
`i, j, k` per un `int`;
`d` per un `double`;
`f` per un `float`;
`o` per un `Object`;
`s` per un `String`;

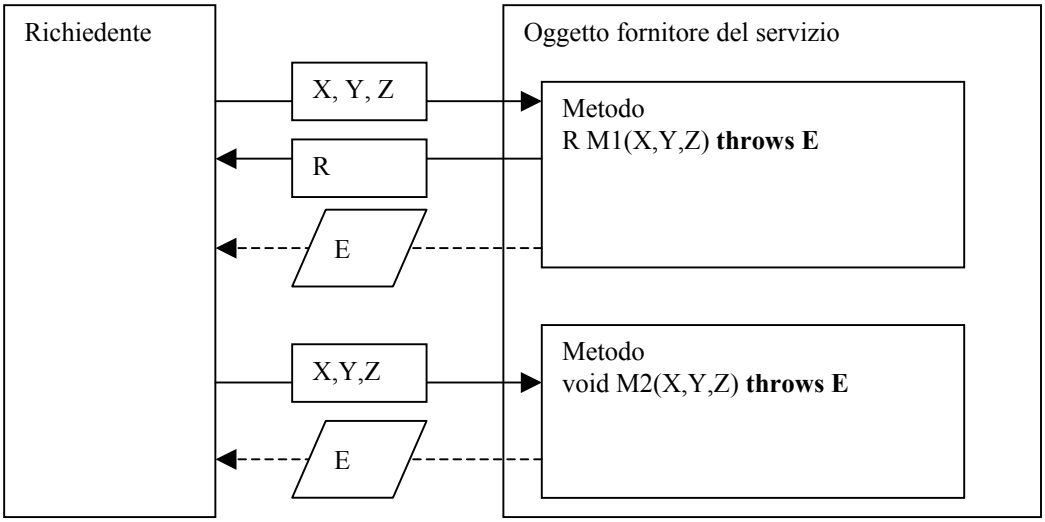
Trattamento delle Eccezioni

La segnatura di un metodo come contratto

La segnatura di un metodo (lista dei tipi dei parametri di input, tipo del risultato se c'è) rappresenta un contratto per la fornitura di un servizio: chi richiede il sevizio deve fornire una lista di valori conformi ai tipi dei parametri input; in cambio otterrà un valore del tipo del risultato, oppure niente se il metodo non ha risultato.



A volte il contratto, come spesso succede per i contratti, prevede delle **eccezioni**. Qualcosa può andare storto, e in tal caso può risultare impossibile fornire il servizio, e viene sollevata un'eccezione.
In un universo ben ordinato, la possibilità di sollevare eccezioni deve essere esplicitamente citata nel contratto, e l'eccezione deve essere chiaramente riconoscibile e gestibile dal richiedente del servizio.
Lo schema disegnato sopra si modifica, per prevedere la restituzione del risultato oppure la comunicazione dell'eccezione.



Nel diagramma sono distinti i flussi normali di informazioni (frecce continue) e in caso di eccezione (frecce tratteggiate).
Supponiamo il caso di un metodo che calcola la radice quadrata di un numero reale.
Il contratto può essere il seguente (segnatura di un metodo):

- `double squareRoot(double x)`

Tuttavia questo contratto non è completo. Può succedere che x sia negativo; in questo caso il fornitore del servizio non può calcolare la radice quadrata di x. Bisogna riportare questa clausola nel contratto:

- `double squareRoot(double x) throws Exception`

Adesso per quanto riguarda il contratto (segnatura) siamo a posto.
Analizziamo più da vicino ciò che succede in casa del fornitore del servizio, e poi in casa del richiedente, per gestire i casi eccezionali.

Il caso del fornitore del servizio

Il fornitore del servizio deve cercare di fornire il servizio nel modo "normale"; se ciò non risulta possibile deve sollevare un'eccezione documentata.

Un esempio di codice:

```
class Fornitore {
      double squareRoot(double x) throws Exception {
            if (x>=0) return Math.sqrt(x);
            else throw new Exception("Il numero " + x + " è negativo");
      }
}
```

Note:

1. Nel contratto è presente la clausola **throws Exception**.
2. Nel modello di calcolo di Java è prevista una azione speciale: quella di sollevare eccezioni. Essa è denotata dalla parola chiave **throw**.
3. **throw** è un operatore che si applica a oggetti di tipo **Throwable**; tra questi ci sono quelli di tipo **Exception** (Exception è un sottotipo di Throwable; Exception è una classe di java.lang).
4. Nel caso si voglia sollevare un'eccezione, visto che si tratta di un'oggetto, bisogna crearlo: quindi si comprende l'uso dell'operatore **new**.

Il caso del richiedente

Potremmo scrivere:

```
class Richiedente {
  void calcolo(double x) {
      ...
      Fornitore f = ...//un'istanza di Fornitore
      double y = f.squareRoot(x);
      System.out.println("Il risultato è " + y);
  }
  ...
}
```

Se facciamo così, otteniamo un errore di compilazione. Infatti non ci stiamo comportando secondo le regole della civile convivenza.

Dobbiamo gestire il caso dell'eccezione.

```
Ecco un esempio in cui l'eccezione è gestita:
class Richiedente {
  void calcolo(double x) {
      ...
      Fornitore f = ...//un'istanza di Fornitore
      try {
        double y = f.squareRoot(x);
        System.out.println("Il risultato è " + y);
      }
      catch (Exception exc) {
        System.out.println("Qualcosa è andato storto: " +
          exc.getMessage());

      }
  }
  ...
}
```

Note:

1. La richiesta di un servizio che può generare eccezioni deve essere effettuata nell'ambito di un blocco **try**.
2. La gestione dell'eccezione è effettuata nell'ambito di un blocco **catch** associato al **try**. Il blocco catch è qualificato con il tipo dell'eccezione (nel nostro caso il tipo generico **Exception** di java.lang) e assegna un nome (nel nostro caso **exc**) all'eventuale eccezione da gestire.

Nel corso di una esecuzione di **calcolo**, il sistema inizierà l'esecuzione del blocco try.

Se l'esecuzione del metodo f.squareRoot non solleva eccezioni, il blocco catch non viene eseguito.

Se invece l'esecuzione del metodo f.squareRoot solleva un'eccezione:

- l'esecuzione del blocco **try** viene interrotta
- viene eseguito il blocco **catch**.

Un richiedente pigro

Non è obbligatorio gestire localmente tutte le eccezioni che nascono nel corso dell'esecuzione di un metodo. Se l'eccezione non viene gestita è necessario segnalare nella segnatura del metodo (in questo caso nel metodo **calcolo**) che nel corso della esecuzione del metodo possono essere sollevate eccezioni.

```
class Richiedente {
  void calcolo(double x) throws Exception {
      ...
      Fornitore f = ...//un'istanza di Fornitore
      double y = f.squareRoot(x);
      System.out.println("Il sisultato è " + y);
  }
  ...
}
```

In questo caso non gestiamo l'eccezione, ma è chiaro a tutto il mondo che può accedere al metodo calcolo che un'eccezione può essere generata nel corso dell'esecuzione di tale metodo.
Infatti anche il metodo **calcolo** rappresenta un servizio, il cui contratto è nella segnatura del metodo. La classe **Richiedente** si comporta come **richiedente** del servizio **squareRoot**, ma come **fornitore** del servizio **calcolo**.
Il servizio calcolo può essere disponibile al pubblico (public void calcolo...), riservato a un'utenza qualificata (protected void calcolo...) oppure riservato per uso interno (private void calcolo...). In tutti i casi le regole sono le medesime:
Se nel corso dell'esecuzione di un metodo può essere generata un'eccezione (da noi stessi o dai nostri fornitori di servizi) è obbligatorio seguire una delle due strade

- gestire l'eccezione, che in tal caso non sarà visibile ai nostri utenti (a coloro che richiedono il nostro servizio);
- segnalare l'eccezione ai nostri utenti inserendo la clausola throws nella nostra segnatura.

Nel caso in cui si decida di trattare l'eccezione mediante un blocco `try-catch`, può accadere che ci siano azioni da eseguire in ogni caso cioè sia nel caso in cui il codice nel blocco `try` sia eseguito senza avere causato alcun problema sia nel caso in cui sia stato eseguito il blocco `catch` perché è stata lanciata l'eccezione. In questa situazione viene scritto in un altro blocco che inizia con la parola chiave `finally`. Il blocco `finally` è opzionale in genere serve per liberare risorse che entrambe le esecuzioni hanno impegnato.

Diverse classi di eccezioni

Abbiamo detto che ogni eccezione è un oggetto, deve pertanto essere esemplare di un'opportuna classe. E' richiesto che ogni eccezione sia un oggetto della classe `Throwable`, o meglio di una delle sue sottoclassi.
`Throwable` è una sottoclasse diretta di `Object` ed ha molte sottoclassi; essa ha due sottoclassi dirette `Error` e `Exception`.
`Error`: i suoi esemplari sono "lanciati" dalla JVM quando accade qualche guaio grave, un programma java non "lancia" oggetti di tipo `Error.`
Quando si decide che è necessario sollevare un'eccezione occorre scegliere se:

- utilizzare `Exception` o una delle sue sottoclassi
- costruire una propria classe che estende una delle sottoclassi di `Exception`; in tal caso è buona norma appendere la stringa `Exception` al nome che si vuole dare alla classe. Si fa ciò quando si ritiene che nessuna delle classi già costruite rappresentino opportunemente il tipo di eccezione da trattare.

La classe `Exception` è una classe molto generale, è sottoclasse di `Throwable` che ha i seguenti costruttori e metodi pubblici:

```
public Throwable()
public Throwable(String s)

//metodi pubblici
public String getMessage()
public String getLocalizedMessage()
public void printStackTrace()
public void printStackTrace(java.io.PrintStream s)
public void printStackTrace(java.io.PrintWriter s)
public native Throwable fillinStackTrace()
```

Nella classe ci sono due costruttori `Throwable()`e `Throwable(String s)`. La stringa di input per il secondo costruttore è un messaggio che l'utente utilizza per fornire una descrizione del problema che causa l'eccezione. Il metodo `getMessage` fornisce il nome qualificato della classe e una breve descrizione dell'eccezione. I tre metodi

`printStackTrace` descrivono lo stato dello stack della Java Virtual Machine nel momento in cui si è verificato l'evento eccezionale.
La classe `Exception` è semplicissima, ha due solo costruttori:

```
public class Exception extends Throwable
public Exception()
public Exception(String s)
```

Le sue sottoclassi indicano vari tipi di eccezioni, per esempio `IllegalAccessException` segnala che un particolare metodo non è stato trovato, mentre `NegativeArraySizeException` indica che un programma ha provato a creare un array con un numero di elementi (dimensione) negativo. Una particolare sottoclasse di `Exception` è di rilievo, è `RunTimeException,` essa rappresenta le eccezioni che possono venire sollevata dalla JVM a runtime.
Come tutte le eccezioni, anche quelle della classe `RunTimeException` si possono raccogliere, ma non è obbligatorio farlo. Il compilatore consente che esse non siano nè raccolte da una `catch` clause, né rilanciate da un `throw` statement. Ciò perché esse sono purtroppo ubiquitarie e un tentativo di trattarle tutte renderebbe il codice illegibile perché troppo complesso. Tra le `RunTimeException` particolarmente interessante è la `NullPointerException` che si ha ogni volta che un metodo cerca di accedere ad un componenete (variabile o metodo) di un oggetto inesistente.

Se in una `catch` clause, l'eccezione raccolta è di tipo `Exception`, essa potrà raccogliere le eccezioni di ognuna delle sottoclassi di tale classe e quindi in particolare anche qualsivoglia tipo di `RunTimeException` (aritmetiche, eccezioni sugli array, sulle stringhe). E' utile pr una strategia di recupero identificare meglio le eccezioni da raccogliere.

Architettura ed eccezioni

Quando si disegna un insieme di classi è importante progettare bene interfacce e classi ma anche le eccezioni che metodi e costruttori devono eventualmente sollevare.
Può essere necessario sollevare anche eccezioni trattate. Si fa ciò mediante un throw statement al termine di una catch clause. Per esempio, se progettiamo e realizziamo un sistema complesso, possiamo pensare che le classi di cui esso è costituito siano strutturate in strati. Abbiamo classi che descrivono il comportamento del sistema, su di esse classi che costituiscono l'applicazione, classi che realizzzano l'interfaccia grafica per l'interazione con l'utente. Le eccezioni trattate ad ogni livello sono diverse. Devono giungere all'interfaccia grafica solo le informazioni per cui l'utente può fare qualche cosa, quindi informazioni che comunicano disattenzioni gravi o errori.

Teoria degli Stream

Comunicazione con il mondo esterno

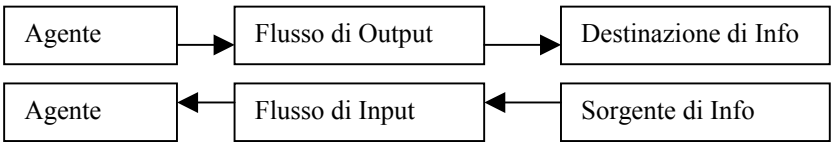
Mediante l'uso della parola comunichiamo con il mondo esterno.
Comunichiamo nei due sensi:

- parliamo
- ascoltiamo.

Parlare e ascoltare

Nel parlare il mondo esterno è visto come **una destinazione di informazioni**. Nell'ascoltare il mondo esterno è visto come **una sorgente di informazioni**. In ambedue i casi il contenuto dell'informazione è vista come una **sequenza di "pacchetti" di informazione**; in questo caso i pacchetti sono i fonemi del linguaggio parlato.
Si tratta di una sequenza di lunghezza che a priori non è determinata. Nell'atto del parlare, possiamo sempre decidere di aggiungere parole fino a che non decidiamo di chiudere il discorso.
Nell'atto dell'ascoltare, non sappiamo quando finirà il discorso finchè il nostro interlocutore non ne comunica la chiusura.

Modello generale



Possiamo formalizzare il modello generale della comunicazione con il mondo esterno con tre (ipotetiche) interfacce:

```
public interface IPacket {} // l'oggetto degli scambi con il mondo esterno
public interface IPacketOutputStream {
  void write(IPacket p) throws Exception; //scrittura di un pacchetto
  void close() throws Exception; // chiusura della comunicazione
}
public interface IPacketInputStream {
  boolean isFinished() throws Exception;
  IPacket read() throws Exception;
  void close() throws Exception;
}
```

L'interfaccia IPacket definisce l'unità di comunicazione (nell'esempio sopra il fonema)
L'interfaccia IPacketOutputStream consente di scrivere un pacchetto alla volta e di chiudere la comunicazione;
L'interfaccia IPacketInputStream consente di

- conoscere se la comunicazione è stata chiusa da parte della sorgente;
- leggere un pacchetto alla volta
- chiudere una comunicazione che non ci interessa più.

Qualcosa può sempre andare storto

Tutti i contratti degli Stream comprendono una clausola che ci mette sull'avviso: qualcosa può andare storto. Si tratta di interagire con il mondo esterno, e possono **sempre** nascere problemi. Nel mondo esterno ci sono i disastri naturali, il deterioramento dei materiali, i nemici, ..., e il caso. Quindi tutti i metodi delle interfacce prevedono **eccezioni**.

Calcolatori e programmi

Il modello si può applicare ai calcolatori, e ai programmi che ne determinano il comportamento. Così accade nell'ambito del mondo **Java**. La comunicazione con il mondo esterno di un agente (in questo caso un qualunque oggetto Java) avviene attraverso la mediazione di appropriati oggetti di flusso (**Stream**).
Siccome la comunicazione può avvenire nei due sensi avremo due tipi di oggetti flusso:

- **InputStream**
- **OutputStream**

Che tipo di pacchetti?

Nel parlare/ascoltare i pacchetti del flusso erano fonemi. Quali sono i pacchetti più adatti per i calcolatori?
Nel calcolatore tutto è codificato mediante numeri interi. E' naturale la scelta in cui ogni pacchetto è un numero intero.
Qualunque intero nella definizione dell'aritmetica?
Troppo scomodo, perchè l'insieme dei numeri interi è illimitato. Sceglieremo gli interi in un intervallo abbastanza comodo, ma che non ci precluda alcuna possibilità.
Un pacchetto particolarmente adatto alla comunicazione tra calcolatori è l'ottetto di bit (una parola di 8 bit). Esso viene comunemente definito **byte**. **Gli Stream sono flussi di byte.**

> NOTA. Il byte di cui stiamo parlando è il tipo primitivo *byte* di java?
> Ricordiamo che il byte di Java è sì un ottetto di bit, ma legato ad una particolare interpretazione: (da -128 a 127).
> La comunicazione è tra comunicazione tra calcolatori qualsiasi, e tra programmi qualsiasi (scritti in qualsiasi linguaggio). Non potremo quindi subordinare l'unità di comunicazione (ottetto di bit) ad una interpretazione particolare.
> L'unità di comunicazione tra calcolatori è il byte astratto, capace di codificare 256 valori diversi. L'unica interpretazione utilizzabile è quella più generale di tutte: un intero da 0 a 255.
> Chiamiamo **byte assoluto** questo concetto per differenziarlo dal tipo primitivo *byte* di java.

Solo numeri interi?

E che cosa accade ai fonemi? li dobbiamo abbandonare? dobbiamo rinunciare a costruire programmi che producono e ricevono fonemi o altre unità di comunicazione?
No. Basterà stabilire delle regole che consentano di codificare i nostri pacchetti in byte, e ricostruire pacchetti a partire da sequenze di byte assoluti

La classe OutputStream

Metodi:

1. public abstract void **write**(int b) throws IOException;
2. public void **write**(byte[] data) throws IOException;
3. public void **write**(byte[] data, int offset, int length) throws IOException;
4. public void **flush**() throws IOException;
5. public void **close**() throws IOException;

Scrittura di un byte

Il metodo (1) è il più importante: ci consente di trasferire un byte (assoluto) al mondo esterno.
Esso utilizza un argomento di tipo **int**, e non **byte**.
Ciò perchè si tratta di byte assoluti, e non byte java. Il byte che vogliamo scrivere può essere il risultato di un calcolo, che fornisce un risultato nell'intervallo 0..255.
Noi forniremo un int (32 bit), e il mondo esterno riceverà solamente gli 8 bit di ordine più basso.
Che cosa succede se forniamo al metodo write un byte di java?

```
byte b = ...
unOutputStream.write(b);
```

Il byte b sarà automaticamente trasformato in un int. Questa trasformazione lascia comunque gli ultimi 8 bit inalterati. Saranno proprio questi ultimi 8 bit a essere scritti sullo stream.

Metodo astratto

Il metodo è astratto: infatti non sappiamo niente di ciò che è dietro il nostro Stream. Ci possono essere molti tipi di codifica, molti tipi di canali trasmissivi, molti tipi di destinazioni. Ci saranno quindi apposite sottoclassi che implementeranno il metodo in modo appropriato.
Avremo molte classi **XXXOutputStream** che tutte forniscono il medesimo servizio astratto di **OutputStream**, ma ciascuna con modalità implementative diverse.

Array di bytes

I metodi 3 e 4 scrivono bytes prelevandoli da array. Il primo scrive un intero array, il secondo ne scrive una parte. Ciascuno di questi byte verrà trasformato in un int prima di essere scritto.

Flush e Close

Il metodo **close**() era stato già postulato dal nostro modello generale. Bisogna che un interlocutore possa segnalare quando ha finito e vuole chiudere. questo metodo libera tutte le risorse impegnate per la comunicazione.
Il metodo **flush**() si comprende se si esaminano gli arricchimenti successivi al modello: dietro il nostro Stream possono nascondersi diversi meccanismi di codifica, o vere e proprie catene di meccanismi, e alcuni dei byte precedentemente inviati potrebbero essere in transito/attesa nelle varie stazioni della catena. Il metodo flush() consente di attivare lo sciacquone e spedire tutto fuori.
Logicamente, close() implica flush() ma non viceversa.

La classe InputStream

E' speculare di OutputStream.

1. public abstract int read() throws IOException;
2. public int read(byte[] data) throws Exception;
3. public int read(byte[] data, int offset, int length) throws IOException;
4. public long skip(long n) throws Exception;
5. public int available() throws IOException;
6. public void close() throws IOException;

Anche qui il metodo più importante è il primo.
il risultato, i di tipo int, può significare due cose:

- se i < 0, (in particolare -1) significa che lo stream è terminato. Non ci sono più bytes.
- se i >= 0, negli ultimi 8 bit troveremo il nostro byte.

Se vogliamo dare agli 8 bit l'interpretazione di un byte java, scriveremo
byte b = (byte) i;
NOTA: lo statement precedente va utilizzato a ragion veduta: esso manterrà i bit, ma non l'interpretazione. Esso trasformerà valori positivi superiori a 127 dell'intero in valori negativi del byte.

Il metodo skip()

Saltare n bytes. Non ci interessano. Il risultato è il numero di bytes effettivamente saltati (<=n).

Il metodo available()

Fornisce il numero di bytes disponibili per essere letti.

Il metodo close()

Libera tutte le risorse impegnate per la comunicazione.

Mark e reset

Alcuni stream supportano un servizio di marcatura che permette di marcare una posizione, leggere alcuni byte dopo la posizione marcata, e poi riprendere a leggere dalla posizione marcata.
Il servizio di Mark e Reset è composto dai metodi:
public boolean markSupported();
public void mark(int readLimit); // si marca la posizione corrente per poter tornare indietro;
public void reset() throws Exception; // si torna all'ultima posizione marcata.
Si può eseguire il reset:

- se markSupported()==true;
- se è stato comandato il mark a una posizione p con un certo readLimit;
- se non si sono letti più byte di readLimit dopo p.

Una procedura di copia

```
...
public void copy(InputStream in, OutputStream out)
throws IOException {
  byte[] buffer = new byte[256];
  while(true) {
```

```
    int count = in.read(buffer);
    if (count<0) break;
    out.write(buffer, 0, count);
  }
}
```
...
E' stato utilizzato un array di 256 bytes per rendere la procedura più efficiente in termini di tempo.
Avere utilizzato il tipo byte di java non ha comportato alcuna perdita di informazione, in quanto non abbiamo effettuato sui byte alcuna interpretazione.

File e File System

Un file è una sequenza di byte memorizzata nel sistema di archiviazione (**file system**) del calcolatore.
I sistemi di archiviazione dei calcolatori possono esere differenti su diversi calcolatori (diversi sistemi operativi). I file system sono indipendenti dai linguaggi di programmazione: in un calcolatore coesistono (e collaborano) programmi scritti in diversi linguaggi).
Tutti i file system si basano su una struttura ad albero:
1. Esiste almeno una **radice** (in Windows esistono tante radici quante sono le unità (*drive*)). La radice è un nodo.
2. Una **Directory** è un nodo del file system; essa possiede un insieme di figli che sono altri nodi.
3. Un **File** è un nodo senza figli, che contiene una sequenza di byte assoluti: i dati del file.

Il file system definisce uno spazio di nomi: la radice e le directory costituiscono contesti per i nomi dei figli.
Un file del file system è individuato da un nome composto:
- C:\programmi\jbuilder\myProjects\myPackage\MyClass.java
- D:\documenti\corsoInformatica\streams.doc

La classe java.io.File

La classe File rappresenta un (possibile) nodo del FileSystem.
Costruttori:
1. **File**(String pathname);
2. **File**(File parent, String child);
3. **File**(String parent, String child);

Un oggetto File non rappresenta un file effettivamente esistente, ma un file potenziale. Infatti esistono metodi per chiedere se esso esiste realmente, e per interrogarne le proprietà:
- boolean **exists**();
- boolean **isFile**();
- boolean **isDirectory**();

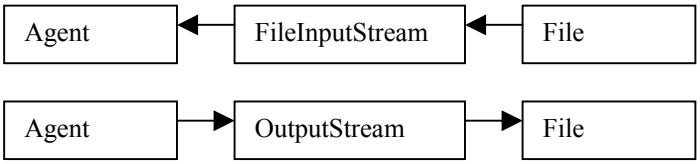
Nel caso in cui il file non esiste, è possibile crearlo vuoto:
- boolean **createNewFile**() throws IOException;

Nel caso in cui esso esiste, è possibile distruggerlo:
- boolean **delete**();

Esistono molti altri metodi per consultarne la dimensione, la data di ultimo aggiornamento, etc; per percorrere l'albero del file system in sù e in giù; etc. Consultare i manuali.

FileInputStream e FileOutputStream

Un file può essere una **destinazione di informazioni** per un *OutputStream* oppure (se preesistente) **una sorgente di informazioni** per un *InputStream*.



La sottoclasse di InputStream che tratta i file è **FileInputStream**; analogamente la sottoclasse di OutputStream che tratta i file è **FileOutputStream**.
Non c'è nulla da dire sul servizio di questi stream che non sia stato detto per le superclassi astratte.
Le uniche cose notevoli sono i costruttori (che non esistevano per la superclasse astratta):
1. **FileInputStream**(File f) throws IOException;
2. **FileInputStream**(String name) throws IOException;

Il secondo costruttore costruisce in un solo colpo File e Stream.

Per FileOutputStream i costruttori sono:
1. **FileOutputStream**(File f) throws IOException;
2. **FileOutputStream**(String fileName) throws IOException;
3. **FileOutputStream**(String name, boolean append) throws IOException;

I primi due costruttori creano il file (vuoto) se non esiste, e lo svuotano se esiste.
Il terzo costruttore può (se append==true) mantenere i vecchi dati, posizionandosi in coda.

Stream di Rete (package java.net)

Abbiamo già parlato, a proposito di nomi e contesti, della convenzione (URL = Uniform Resource Locator) che permette di identificare qualsiasi file contenuto in un calcolatore (host) connesso alla rete.
Un URL è strettamente associato con un nome composto, come:

- http://www.cnn.com/today/news/electionsUS.html

Il formato è *protocol://hostName/path*
Come si vede, ci troviamo in un contesto molto simile a quello del file system di una macchina: è solamente più grande. Abbiamo messo insieme tutti i file system di tutte le macchine connesse alla rete, anche se gestiti ciascuno da un particolare sistema operativo.

E' logicamente semplice estendere i discorsi fatti prima a proposito dei file.
Come prima potevo costruire un file

```
File aFile = new File(aPath);
```

così, nel contesto della rete, posso costruire un oggetto URL, ed una connessione ad esso associata:

```
URL aUrl = new URL(aNetworkPath);
URLConnection con = aUrl.openConnection();
```

Prima potevo costruire un InputStream su un file con lo statement:

```
InputStream x = new FileInputStream(aFile);
```

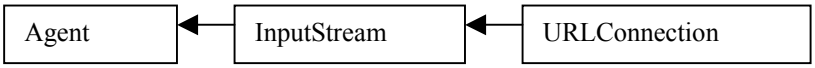
Analogamente, nel contesto della rete, posso ottenere un InputStream con uno statement leggermente differente dal punto di vista sintattico, ma strettamente simile dal punto di vista semantico:

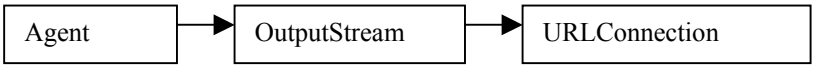
```
InputStream y = con.getInputStream();
```

A questo punto gli InputStream x e y forniscono lo stesso servizio:

- l'uno considerando come sorgente di informazione un file del file system locale;
- l'altro considerando come sorgente di informazione un file di un calcolatore remoto, con sistema operativo sconosciuto.

Chiaramente il macchinario sottostante impiegato nei due casi è molto diverso, ma il servizio è uguale. Potenza dell'astrazione.
Ecco il modello nel caso della rete:





Logicamente la connessione di rete (URLConnection) può fornire anche un OutputStream: in questo modo possiamo inviare informazioni ad un calcolatore remoto.

Come per tutte le estensioni degne di rispetto, il contesto esteso comprende il contesto di partenza come caso particolare.
Possiamo costruire un URL a partire dalla stringa

```
file://aPath (NOTA: il protocollo è "file" al posto di "http")
```

In questo caso abbiamo costruito una connessione di rete al nostro stesso file system. Questa connessione è in grado di fornirci InputStream e OutputStream del tutto equivalenti a quelli prima costruiti con gli oggetti File.

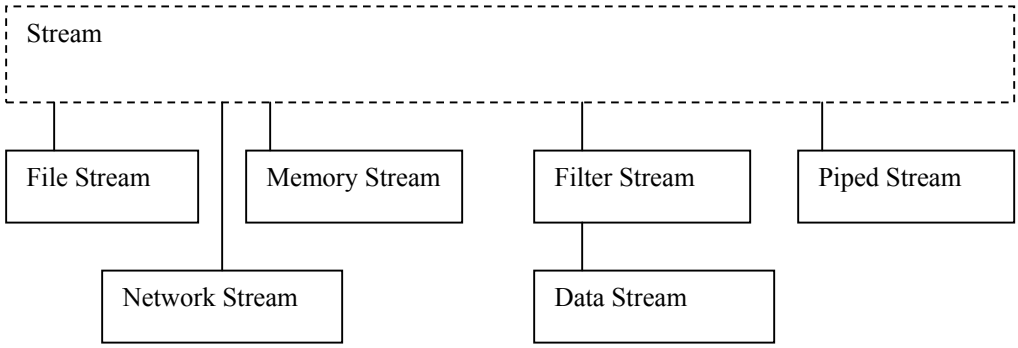
Dietro le quinte

La semplicità logica e l'eleganza concettuale delle astrazioni non devono farci dimenticare le specificità delle tecnologie coinvolte nei due casi.

- Nel caso del file system abbiamo a che fare con la specificità dell'hardware e del sistema operativo locale, con le loro soluzioni particolari.
- Nel caso delle connessioni di rete, sono coinvolte componenti tecnologiche di trasmissione e di intermediazione. Inoltre devono essere riportate ad un contesto unitario e standard particolarità delle diverse CPU e dei diversi

sistemi operativi. Basti pensare che le diverse CPU presenti sul mercato possono avere (ciascuna a casa propria) particolarità differenti nell'ordinamento dei bit nei byte delle parole di memoria.

Sottoclassi di InputStream e OutputStream



Buffered Stream

I Buffered Stream sono stream che hanno come fonte o destinazione dei dati altri stream. Essi aggiungono un servizio di bufferizzazione.
BufferedInputStream e **BufferedOutputStream**.

Data Stream

- DataInputStream e DataOutputStream consentono di leggere/scrivere i tipi primitivi di java e le stringhe. Attenzione: i dati sono java-oriented!
- ObjectInputStream e ObjectOutputStream consentono di leggere/scrivere interi oggetti (Serializzazione).

Mostri

PrintStream: scrive caratteri come bytes. E' frutto di un errore di gioventù di Java. Rimane perchè System.out è nato come PrintStream.

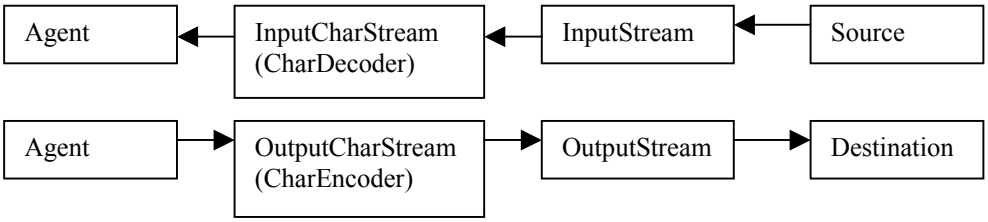
Memory Stream

Sono stream che hanno come fonte o destinazione dei dati degli array di byte.
ByteArrayInputStream e ByteArrayOutputStream

Caratteri e Testi

In apertura del discorso sugli Stream abbiamo fatto l'esempio della parola e dei fonemi. In quell'esempio i flussi (Stream) erano flussi di fonemi.
Poi abbiamo parlato degli Stream di Java, che sono flussi di byte (il byte internazionale 0.255 e non il byte particolare di java -128..127).
Abbiamo anticipato che attraverso i flussi di byte possono essere trasmessi anche flussi di pacchetti di altro tipo, come i fonemi; e abbiamo ipotizzato la presenza di componenti ausiliarie (Encoder/Decoder) dietro gli Stream.
Mettiamo ora alla prova la nostra architettura ponendoci il problema della comunicazione di testi.
Testo ::= Flusso di Caratteri
Carattere ::= carattere UNICODE

Il nostro agente vuole trattare sequenze di caratteri. Esso desidererebbe qualcosa come un InputCharacterStream e un OutputCharacterStream. Purtroppo il mondo esterno (file system, canali trasmissivi) sono solo capaci di trattare i byte. Abbiamo bisogno di un modello del tipo:



Qualcuno potrenbbe obiettare che, visto che un carattere UNICODE corrisponde alla stessa quantità di informazione contenuta in due bytes, encoder e decoder sono così semplici da potere essere trascurati e (quasi) eliminati. Basta considerare ciascun carattere come una coppia di byte.
In pratica sono state stabilite convenzioni che utilizzano paricolari codifiche per i caratteri UNICODE al fine di ottimizzare la trasmissione delle informazioni. In base a queste convenzioni un carattere unicode non viene codificato rigidamente in due byte, ma vengono utilizzate codifiche a lunghezza variabile. Queste codifiche tengono conto delle probabilità di occorrenza dei vari caratteri, in modo da codificare in un solo byte i caratteri a probabilità maggiore, e in due o tre byte i caratteri a probabilità minore.
Se queste sono le convenzioni della rete, dobbiamo adeguarci. Da qui la necessità delle particolari componenti che nel diagramma sono chiamate

- InputCharStream (Decoder)
- OutputCharStream (Encoder)

Essi nel linguaggio Java si chiamano più semplicemente **Reader** e **Writer**.

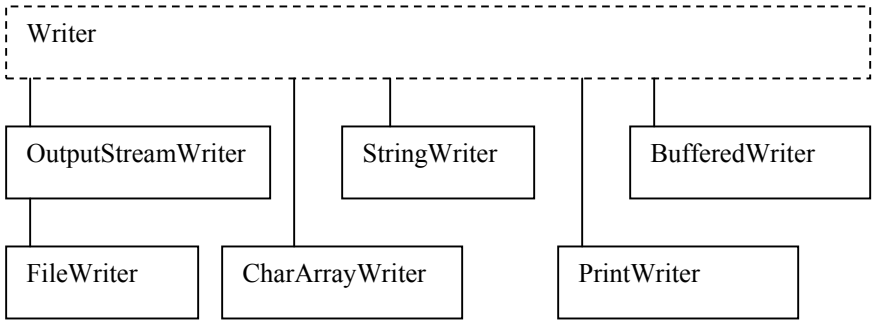
La classe Writer (astratta)

1. public void write(int c) throws IOException;
2. public void write(char[] text) throws IOException;
3. public void write(String s) throws IOException;
4. public abstract void write(String s, int offset, int length) throws IOException;
5. public abstract void flush() throws IOException;
6. public abstract void close() throws IOException;

Il servizio è chiaro: scrittura di un carattere, una stringa, un array di caratteri.
Un Writer si appoggia su un OutputStream o su qualche meccanismo di memorizzazione (char array o StringBuffer). Il suo compito principale è quello di codificare i caratteri in byte.

Implementazioni di Writer

La classe Writer è astratta, e quindi non istanziabile. Sono istanziabili invece le sue sottoclassi concrete, che si differenziano per la destinazione dei dati, o per l'arricchimento del servizio.
I costruttori delle classi concrete accettano come parametro il nome di una regola di codifica (encoding).
Infatti esistono diverse modalità di codifica dei caratteri in byte.



Sottoclasse di Writer	*Destinazione*
OutputStreamWriter	OutputStream
FileWriter	File
StringWriter	StringBuffer
CharArrayWriter	char[]

BufferedWriter	Writer
PrintWriter	Writer, OutputStream

Destinazione dei dati

OutputStreamWriter si costruisce su un OutputStream preesistente.
FileWriter è una composizione di OutputStreamwriter e FileOutputStream.
StringWriter assume come destinazione di dati uno StringBuffer (a cui si può chiedere il risultato come stringa).
CharArrayWriter mantiene il risultato come array di caratteri.

Arricchimento del servizio

BufferedWriter fornisce il servizio di Writer con l'aggiunta di un buffer, che rende più efficienti le operazioni.
PrintWriter aggiunge al servizio di Writer un insieme di metodi print(...) e println(...) che accettano un argomento di tipo primitivo o Object.

- **print**(x) scrive la rappresentazione di x come stringa;
- **println**(x) scrive la rappresentazione di x come stringa, seguita dal separatore di riga.

Il separatore di riga dipende dalla piattaforma (generalmente \r\n).

La classe Reader

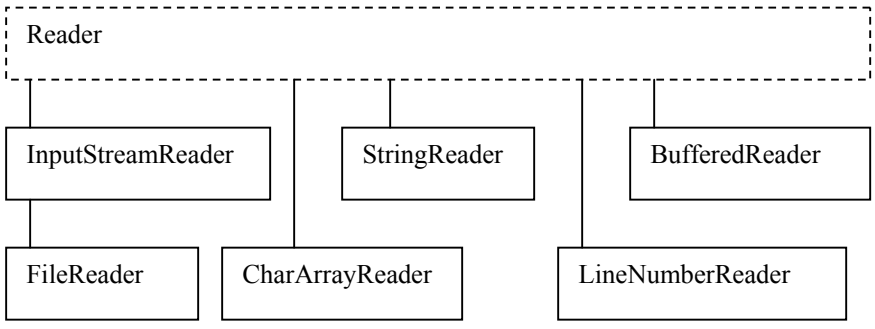
E' speculare alla classe Writer.

- public abstract int **read**(char[] buffer, int offset, int length) throws IOException; lettura in buffer a partire da offset per una certa lunghezza massima length; fornisce il numero di caratteri letti oppure -1 (end of data);
- public int **read**(char[] buffer) throws IOException; come sopra con offset=0 e lunghezza massima pari alla lunghezza di buffer;
- public int read() throws IOException; lettura di un singolo carattere; fornisce il carattere letto oppure -1 (end of data);
- public long skip(long n) throws Exception; salta max n caratteri; fornisce il numero effettivo dei caratteri saltati.
- public boolen ready() throws Exception; dichiara se ci sono caratteri da leggere.

Un Writer offre un servizio di Mark & Reset se il meccanismo sottostante lo consente.

Implementazioni di Reader

La classe Reader è astratta, e quindi non istanziabile. Sono istanziabili invece le sue sottoclassi concrete, che si differenziano per la fonte dei dati, o per l'arricchimento del servizio.
I costruttori delle classi concrete accettano come parametro il nome di una regola di codifica (encoding).
Infatti esistono diverse modalità di codifica dei caratteri in byte.



Sottoclasse di Reader	*Fonte*
InputStreamReader	InputStream
FileReader	File
StringReader	String
CharArrayReader	char[]
BufferedReader	un altro Reader
LineNumberReader	un altro Reader

Fonte dei dati

InputStreamReader si costruisce su un InputStream preesistente.
FileReader è una composizione di InputStreamReader e FileInputStream.

StringReader assume come fonte di dati una Stringa.
CharArrayReader assume come fonte un array di caratteri.

Arricchimento del servizio

BufferedReader fornisce il servizio di Reader con l'aggiunta di un buffer, che rende più efficienti le operazioni.
LineNumberReader aggiunge al servizio di Reader due metodi per la gestione di una numerazione di righe

- int **getLineNumber**() fornisce un numero progressivo di riga
- void **setLineNumber**(int k) imposta il numero di riga iniziale della progressione.
- String **readLine**() throws IOException fornisce una riga.

Per tenere conto delle righe viene utilizzato un separatore di riga che dipende dalla piattaforma (generalmente \r\n).

Collezioni

Le collezioni sono trattate in modo sofisticato e completo nel package java.util. Tuttavia impadronirsi dei concetti e delle tecniche delle classi di java.util non è facile, in quanto si tratta di concetti e tecniche non banali. Quella che segue è una trattazione semplificata che può costituire una introduzione alle classi di java.util.
Il problema delle collezioni è molto vasto, ed è stato oggetto di studio fin dai primordi dell'informatica. Per esplorarlo occorre quindi un metodo.
Seguiremo il seguente:

1. Definire un certo numero di servizi che potremo richiedere alle collezioni.
2. Analizzare un certo numero di meccanismi che possono essere utilizzati nell'implementazione delle collezioni.
3. Discutere la adeguatezza dei vari meccanismi per la implementazione dei vari servizi.
4. Effettuare a questo punto una panoramica sintetica delle classi di java.util.

Il concetto astratto di collezione

Una collezione è un **contenitore**: un oggetto che contiene un certo numero di altri oggetti (**elementi**).
Un contenitore di elementi possiede una **cardinalità** n >= 0.
Una collezione di cardinalità n contiene esattamente n elementi.
Se la cardinalità è 0, la collezione si dice **vuota**.
Servizio:

- consultazione degli elementi
- aggiunta di nuovi elementi
- rimozione di elementi

Le modalità di aggiunta/rimozione e di accesso agli elementi possono essere diversificati per tipi diversi di contenitore.

Servizio minimo: IContainer

```
public interface ICollection {
  public boolean isEmpty();
  public int size();
}
```

Stack (pila)

Lo stack è una sequenza di elementi in cui gli elementi si aggiungono in coda e si prelevano dalla coda.
LIFO: last in, first out.

IStack

```
public interface IStack extends IContainer {
  public void push(Object v); // size = size + 1
  public Object top(); //interroga la cima (l'ultimo)
  public Object pop(); //estrae l'ultimo: size = size - 1
}
```

Queue (coda)

La coda è una sequenza di elementi in cui gli elementi si possono aggiungere solo in coda, e si possono prelevare solo dalla testa.
FIFO: first in, first out.

IQueue

```
public interface IQueue extends IContainer {
  public void put(Object v); // aggiunge in coda; size = size + 1
  public Object get(); // preleva dalla testa; size = size - 1
}
```

Iterator

Una collezione può offrire un servizio di consultazione di tipo Iterator.

Il servizio Iterator è costruito sul concetto di successore (next).
Il modello è analogo a quello utilizzato da Peano per la definizione dei numeri interi. Peano definisce il dominio dei numeri interi attraverso due concetti: *zero* e *successore*. Per Peano:

- esiste il numero *zero*, che non è il successore di nessun altro numero;
- dato un numero, esiste il *successore* di quel numero.

Applicando il modello di Peano, avremmo per le collezioni:

- un primo elemento
- il successore di un qualunque elemento.

Ai nostri fini il modello di Peano deve essere indebolito, perchè una collezione può essere vuota. Il modello diventa:

- può esistere un primo elemento (se la collezione è non vuota);
- dato un elemento, può esistere il successore.

A questo punto, il concetto di primo elemento (lo zero di Peano) può essere eliminato, perchè non se ne può più postulare l'esistenza. Resta solo la funzione successore, che può essere applicabile oppure no.

```
public interface Iterator {
  boolean hasNext(); // c'è un successore?
  Object next(); // il successore
}
```

Per iterare su una collezione iterabile la procedura è la seguente:

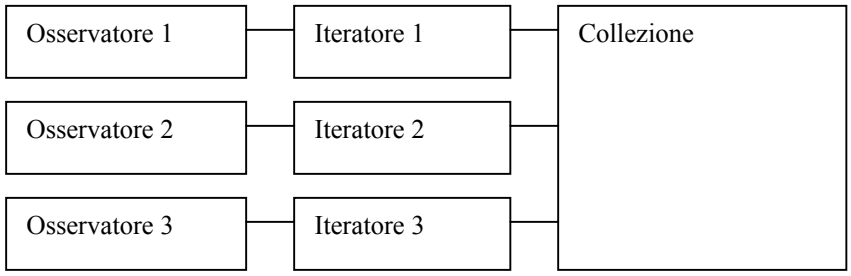
```
Iterator iter = ...
while (iter.hasNext()) {
  Object element = iter.next();
  .... trattamento dell'elemento ...
}
```

Iterabile

Il concetto di Iterator non è sufficiente per rendere una collezione iterabile. Infatti la stessa collezione può essere iterata in maniera indipendente da parte di più osservatori. Ogni osservatore effettua una diversa iterazione (utilizza un diverso iteratore). Una collezione iterabile deve essere in grado di fornire iteratori diversi a diversi osservatori:

```
public interface Iterable {
  Iterator getIterator(); // crea un nuovo iteratore
}
```

Un oggetto di tipo Iterable è una collezione iterabile.
Un oggetto di tipo Iterator è un iteratore su una collezione.



Stack e code non hanno bisogno di iterabilità.

Invalidazione degli iteratori

Supponiamo che una collezione iterabile abbia fornito n iteratori a n osservatori, e poi venga modificata (ad esempio mediante l'aggiunta o la rimozione di un elemento). In questo caso bisogna rendere non più utilizzabili gli iteratori forniti in precedenza. Come ottenere questo risultato? Gli iteratori forniti potrebbero essere molti; non sappiamo che fine essi abbiano fatto, nè se essi siano tuttora in uso.
La soluzione standard è la seguente:

- La collezione iterabile mantiene una variabile privata intera **updateCount** (contatore degli aggiornamenti).
- La collezione incrementa updateCount ad ogni operazione di modifica.
- Quando un iteratore viene creato, esso registra nel proprio stato il contatore degli aggiornamenti della collezione.
- Ogni volta che l'iteratore agisce sulla collezione, esso controlla la corrispondenza del proprio contatore con quello della collezione. Se la corrispondenza non c'è, viene sollevato un errore di "Modifica concorrente".

ICollection

Un contenitore che è iterabile ed è in grado di dire se contiene un oggetto "uguale" ad un oggetto dato.

Uguale in che senso? nel senso di **equals**.

```
public interface ICollection extends IContainer implements Iterable {
  boolean contains(Object v);
  boolean add(Object v);
  boolean remove(Object v);
  (*) boolean containsAll(ICollection c);
  (*) void addAll(ICollection c);
  (*) void removeAll(ICollection c);
}
```

Il servizio **contains**(...) richiede ad una collezione di stabilire se essa contiene un oggetto uguale a un oggetto dato. In linea di principio la richiesta è pienamente realizzabile: infatti il servizio di iterabilità garantisce che si possa iterare su tutti gli oggetti, e il metodo equals(...) degli elementi garantisce la realizzabilità del riconoscimento.
Tuttavia l'iterazione è un'operazione lunga. Vedremo che alcuni sottotipi di collezioni dovranno essere ottimizzati con meccanismi particolari per rispondere in modo efficiente a questa richiesta.

Il metodo **remove**(Object v) implica anch'esso la ricerca di un elemento uguale a v, e quindi pone gli stessi problemi di ottimizzazione di contains(...).

I metodi segnati con (*) non sono altro che una composizione dei metodi semplici precedenti con una iterazione sulla seconda collezione.
Se la collezione non è dotata di alcun meccanismo per velocizzare il metodo contains(), il metodo containsAll richiederà non una, ma n iterazioni: una per ciascun elemento della seconda collezione c.

Set

Un insieme è una collezione che non contiene elementi duplicati. Ciò significa che ogni operazione add(...) implica l'esecuzione interna di una operazione contains(...). Quindi i set devono essere implementati con meccanismi che ottimizzano l'operazione contains(...)

```
public interface ISet extends ICollection {}
```

Si tratta di un'interfaccia vuota, il cui ruolo è puramente quello di segnalare che non ci sono elementi duplicati (è una cosiddetta **marker** interface).

Liste

Una lista è una collezione che mantiene una sequenza di elementi. Ad ogni elemento è assegnato un indice 0..size-1 che indica la sua posizione nella sequenza. Si può accedere a un qualunque elemento in base al suo indice. Si possono aggiungere/togliere elementi in testa, in coda, in qualunque posizione.

```
public interface IList extends ICollection {
  public void add(int index, Object v);
  public Object get(int index);
  public Object remove(int index);
  public ind indexOf(Object v);
  public void set(int index, Object v); // sostituisce
}
```

Come si vede, sono stati aggiunti metodi che utilizzano l'indice.
Il metodo indexOf(Object v) fornisce l'indice di un oggetto (-1 se l'oggetto non è presente nella lista).
Il metodo set(...) è una novità concettuale. Senza indici, per sostituire un oggetto x con un oggetto y sarebbe stato necessario utilizzare due operazioni:

- remove(x);
- add(y);

Iterazione e accesso per indice

L'interfaccia IList consente di iterare su tutti gli elementi della collezione senza richiedere un iteratore. Infatti l'osservatore può scrivere:

```
int n = aList.size();
for (int i=0; i<n; i++) {
  Object element = aIndexed.get(i);
  ... tratta elemento ...
}
```

In questo modo tuttavia non si avrebbe il beneficio della invalidazione automatica dell'iteratore in caso di modifica della collezione. Ciò può essere pericoloso in un ambiente con processi concorrenti.

Mappe

Una mappa è un contenitore in grado di registrare associazioni chiave/valore. Non sono ammesse chiavi duplicate: se si registra prima il valore x con chiave k, e poi il valore y con la stessa chiave k, y sostituisce x.
E' possibile ritrovare un valore in base alla sua chiave.
Il concetto di Mappa realizza esattamente il concetto di funzione da un dominio di chiavi a un dominio di valori.
Mappa : Chiave → Valore

```
public interface ISimpleMap extends IContainer {
  void put(Object key, Object element);
  Object get(Object key);
}
```

Una mappa è una collezione?

Una mappa può essere pensata come una collezione (anzi un insieme) di coppie chiave/valore, con il vincolo che non possono esistere due coppie con la stessa chiave. Possiamo chiamare **entrySet** questo insieme.
Essa contiene un insieme di chiavi: **keySet**.
Essa contiene una collezione di valori: **values**;
Possiamo rendere espliciti questi concetti arricchendo l'interfaccia ISimpleMap:

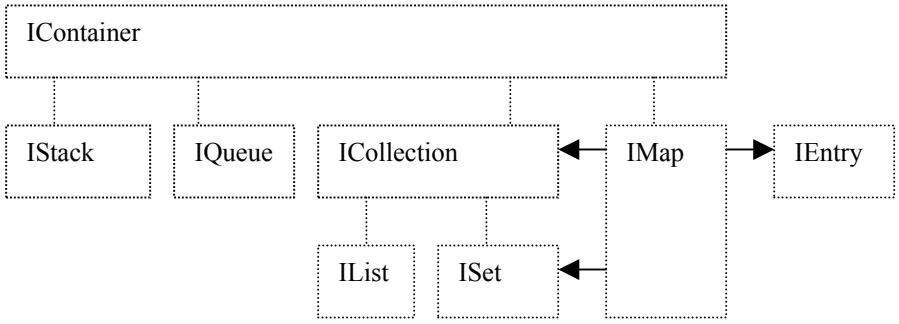
```
public interface IMap extends ISimpleMap {
  public ISet entrySet();
  public ISet keySet();
  public ICollection values();
  boolean containsKey(Object key); // abbreviazione per keySet().contains(key)
  boolean containsValue(Object value); // abbreviazione per
values().contains(value)
}
```

Il metodo entrySet() ci costringe a definire una interfaccia per la coppia chiave/valore:

```
public interface IEntry {
  Object getKey();
  Object getValue();
  void setValue(Object v);
}
```

Gerarchia di servizi

Abbiamo costruito una gerarchia di servizi, a cui corrisponde una gerarchia di tipi (interfacce).



Meccanismi di Implementazione

Abbandoniamo adesso il punto di vista dell'architettura dei servizi, adottando invece il punto di vista dell'ingegneria delle componenti.
Analizzeremo alcuni meccanismi che possono essere utilizzati per implementare i servizi sopra analizzati:

1. l'array dinamico
2. la catena di nodi
3. l'albero binario
4. la tavola di hash

Array dinamico

Il tipo array già presente in Java sarebbe ideale per realizzare le collezioni, se non avesse il difetto di avere una cardinalità fissa, mentre la cardinalità di una collezione cambia con l'aggiunta e l'eliminazione di elementi. Risulta quindi naturale ricercare la realizzazione di un array dinamico.

<table>
<tr><td colspan="2">Array (capacity)</td></tr>
<tr><td>Parte occupata dalla collezione (size)</td><td>Parte libera</td></tr>
</table>

Possiamo immaginare un array con una certa capacità iniziale i cui elementi vengono man mano occupati dagli elementi della collezione.
Tutto va bene finché la cardinalità della collezione ospitata si mantiene minore o uguale alla capacità dell'array.
E se la collezione cresce ancora? si sostituisce l'array originale con uno più grande (ad esempio doppio) e si ricomincia.

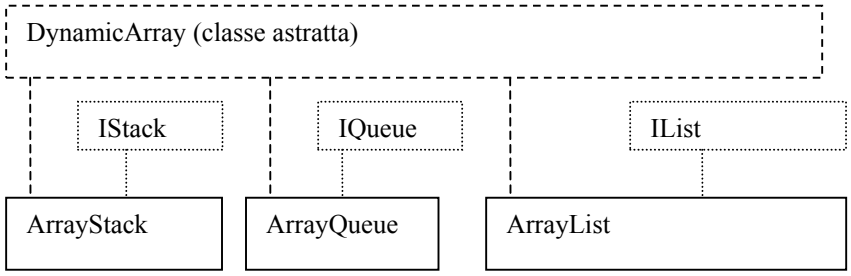
La classe DynamicArray

Ecco un possibile scheletro per la classe DynamicArray:

```
public abstract class DynamicArray {
  private Object[] content;
  private int size;
  //public
  public DynamicArray(int capacity);
  public DynamicArray();
  public int size();
  public boolean isEmpty();
  //protected
  protected void ensureCapacity(int requiredCapacity);
  protected Object get(int i);
  protected void set(Object v, int i);
  protected void add(Object v);
  protected void add(Object v, int i);
  protected Object remove(int i);
  protected Iterator getIterator();
  private class DynamicArrayIterator implements Iterator {
    ...
  }
}
```

- La classe è **astratta**. Infatti non è stata pensata per essere messa a disposizione del pubblico, ma per fornire un semilavorato solido e sicuro per le successive implementazioni.
- La classe possiede pochissimi metodi pubblici. Quasi tutti i metodi che forniscono un servizio sono **protetti** (addirittura tutti i metodi potrebbero essere protetti). Ciò consente a ciascuna delle classi pubbliche che implementeranno i servizi astratti definiti in precedenza (Stack, Coda, Lista, ...) di ritagliare con precisione i metodi da rendere pubblici.
- La classe definisce una classe interna privata DynamicArrayIterator che è un **iteratore** specializzato per questa struttura di dati. Le istanze di questa classe verranno create dinamicamente e fornite agli osservatori che richiedono un iteratore. Gli osservatori si ritroveranno a disposizione oggetti di cui non conoscono la classe (per loro completamente inaccessibile), ma solamente l'interfaccia **Iterator**.

A questo punto è semplicissimo produrre le classi pubbliche (e concrete) che realizzano i servizi definiti prima.



La classe ArrayStack

La classe ArrayStack rende pubblico un servizio di tipo Stack e nasconde tutti gli altri possibili servizi.

```
public class ArrayStack extends DynamicArray implements IStack {
  public ArrayStack() {}
  public void push(Object v) {super.add(v);}
  public Object top() {return get(size()-1);}
  public Object pop() {return remove(size()-1);}
}
```

La classe ArrayQueue

La classe ArrayQueue rende pubblico un servizio di tipo Queue e nasconde tutti gli altri possibili servizi.

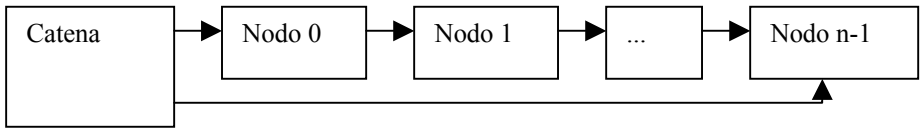
```
public class ArrayQueue extends DynamicArray implements IQueue {
  public ArrayQueue() {}
  public void put(Object v) {super.add(v);}
  public Object get() {return super.remove(0);}
}
```

La classe ArrayList

```
public class ArrayList extends DynamicArray implements Indexed, Iterable {
  public ArrayList() {}
  public Object get(int i) {return super.get(i);}
  public void add(Object v) {super.add(v);}
  public void add(int I, Object v) {super.add(v, i);}
  public Object remove(int i) {return super.remove(i);}
  ...
  public Iterator getIterator() {return super.getIterator();}
}
```

Catena

Una catena è una struttura dinamica costituita da una sequenza di nodi che si richiamano l'uno con l'altro.



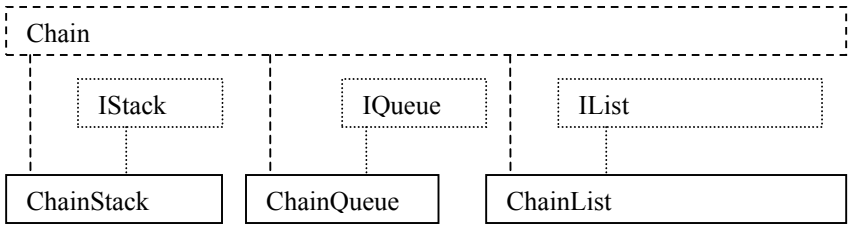
Lo scheletro strutturale della classe è il seguente:

```
public abstract class Chain {
  private ChainNode first, last;
  private int size;
  ...
  private class ChainNode {
    ChainNode next;
    Object element;
  }
}
```

I metodi della classe possono avere la stessa segnatura di quelli di DynamicArray (e naturalmente implementazione del tutto di versa, perchè è diversa la filosofia della struttura dati).

.

A questo punto è semplicissimo produrre le classi pubbliche (e concrete) che realizzano i nostri servizi utilizzando il semilavorato Chain.



Qualcosa per ISet e IMap

ISet e IMap pongono il problema della **ricerca efficiente** di un oggetto all'interno di un insieme. In ISet ciò è necessario per evitare i duplicati; in IMap è necessario per il funzionamento della mappa.
Esamineremo tre soluzioni al problema della ricerca efficiente:

1. Array ordinati
2. Alberi binari
3. Funzioni di hashing

Array ordinati

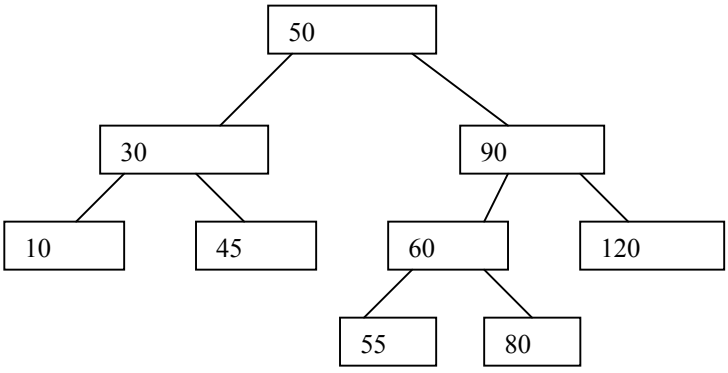
Se un array è ordinato, è possibile esaminare per primo l'elemento di mezzo dell'array (indice = size/2).
Se esso è minore di quello cercato, si potrà acartare la prima metà dell'array; se esso è maggiore, si scarterà la seconda metà.
La ricerca su mezzo array si effettua con la stessa logica, cercando ancora di escluderne metà (cioè un quarto dell'intero array); e così via.
Questo algoritmo si chiama ricerca dicotomica.
Vantaggi: la ricerca è veloce (numero massimo di comparazioni ≈ log2(size)).
Svantaggi: l'operazione di inserimento è più lenta, in quanto ogni elemento deve essere inserito nella posizione giusta, spostando gli elementi superiori.

Alberi binari

Il costo all'inserimento si può ridurre utilizzzando una struttura più agevolmente modificabile: l'albero binario.

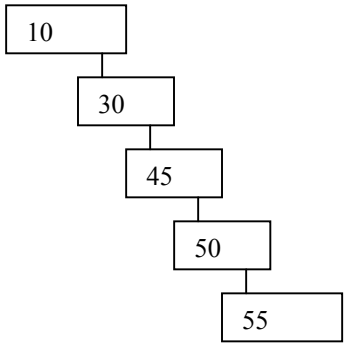
```
class BynaryTree {
  BynaryNode root;
}
class BynaryNode {
  Object element;
  BynaryNode left, right;
}
```

Come si vede, si tratta di una struttura che assomiglia molto alla catena di nodi vista sopra.
Il nodo della catena puntava a un singolo nodo successore. Il nodo dell'albero binario punta a due "successori" o figli possibili: quello di sinistra e quello di destra.
La struttura si presta molto bene a rappresentare una relazione di ordinamento: in tal caso il figlio di sinistra è minore, il figlio di destra è maggiore.



Ricerca: si parte dalla radice. Se l'elemento cercato è minore dell'elemento del nodo, si cerca a sinistra. Se invece è maggiore, si cerca a destra. Se è uguale, la ricerca è finita con successo. Se non ci sono più nodi, la ricerca è fallita.

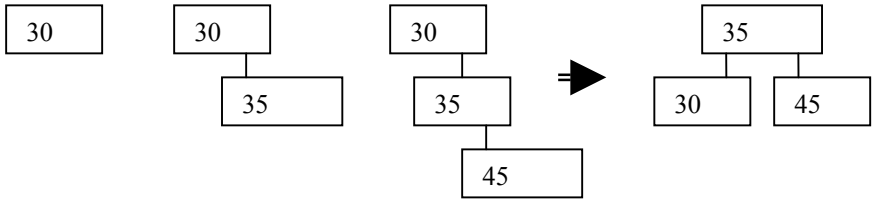
Inserimento: si effettua la ricerca (che fallisce in un certo nodo). La ricerca è fallita in quel nodo perchè mancava la parte di destra (o, rispettivamente, la parte di sinistra). L'inserimento crea appunto questa continuazione a destra (o a sinistra).
Eliminazione: si effettua la ricerca, e si trova il nodo da eliminare. Bisogna eliminare il nodo senza perdere i collegamenti: i fili che rimarrebbero spezzati vanno riannodati. Per esempio, se si deve eliminare il nodo 60 della figura precedente, bisogna garantire che i nodi 55 e 80 possano essere ancora accesssibili a partire dal nodo 90.
Gli algoritmi di ricerca, aggiornamento, cancellazione in un albero binario "normale" sono semplici ed eleganti.
C'è un unico neo: costruendo l'albero mediante inserimenti successivi, l'albero può risultare, nei casi più disgraziati, "sbilanciato".
Esempio: si pensi alla sequenza di inserimenti 10, 30, 45, 50, 55, ...
Senza particolari accorgimenti, nasce l'albero:



in cui cresce, di ogni nodo, solo la parte di destra (e non quella di sinistra). Avrebbe voluto essere un albero binario, e si è sviluppato come una catena. Il costo della ricerca è pari a quello della ricerca in una catena.

Alberi binari bilanciati

Per rimediare a questo problema, sono stati inventati gli alberi binari (auto-)bilanciati. Essi sono alberi binari in cui, in fase di inserimento o di rimozione, si adottano opportuni accorgimenti che "correggono" lo sviluppo di un albero che rischia di sbilanciarsi.
La figura che segue rappresenta in forma molto semplificata come potrebbe crescere un albero bilanciato.



In un albero bilanciato, le operazioni di inserimento e rimozione possono ristrutturare l'albero.
Esistono diversi tipi di alberi binari bilanciati. I costruttori di Java hanno scelto gli alberi rosso-neri. In essi a ciascun nodo è associato un colore (rosso o nero) con regole appropriate. La presenza di due nodi consecutivi rossi provoca la ristrutturazione di un pezzo di albero.

Funzioni di hashing

Un oggetto appartiene al dominio o spazio degli oggetti.
Nella nostra collezione abbiamo un dominio o spazio delle posizioni. Immaginiamo che lo spazio delle posizioni sia suddiviso in un numero intero N di celle.
Il dominio **pos** delle posizioni è l'intervallo 0..N-1.
Immaginiamo disponibile una funzione facilmente calcolabile (funzione di **hashing**) che permette di passare dal dominio degli oggetti al dominio delle posizioni:

hash : Object → pos

La magia si è realizzata: ad ogni oggetto arbitrario siamo in grado di assegnare una "posizione naturale" nella nostra collezione:

- nella fase di inserimento sappiamo dove (in quale cella) collocarlo;
- nella fase di ricerca sappiamo dove (in quale cella) cercarlo.

Problema delle collisioni: il dominio delle posizioni è più piccolo del dominio degli oggetti possibili (la nostra collezione è più piccola dell'universo). Quindi la funzione di hashing può fornire la stessa posizione per più oggetti. Per rimediare a questo inconveniente dobbiamo:

- studiare un modo per far coesistere più oggetti nella stessa cella (ad esempio utilizzando il meccanismo della catena);
- aumentare il numero delle celle (e ricollocare gli oggetti nelle nuove celle) quando si supera una certa soglia di affollamento.

La funzione di hashing è costruita sulla funzione

```
int hashCode()
```

definita per la classe `Object` e ridefinibile in ciascuna classe.
La funzione hashCode() deve avere in ogni classe comportamento coerente con quello della funzione equals(): infatti se due chiavi tra di loro equals danno luogo a valori di hashCode() diversi ci ritroviamo nei pasticci. La funzione hashCode è già coerente con equals in tutte le classi Java di utilizzo universale (String, Integer, Double, ...)

Meccanismi e servizi

I due servizi Stack e Queue si ottengono da qualunque meccanismo adatto a fornire il servizio di Lista. Per essi il problema è quello di nascondere i servizi in più.
Basta qui esaminare la adeguatezza dei vari meccanismi a fornire i tre servizi Lista, Set e Mappa.

Dynamic Array

E' il meccanismo ideale per le liste. L'unico punto debole è il servizio di contains(), ottenibile solo per iterazione.
Per il Set diventa determinante il servizio di contains(); quindi DynamicArray non è una soluzione ottimale per il servizio Set.
Per la Mappa si può pensare a realizzare un array di coppie; ma rimane la debolezza del contains. Non ottimale.

Catena

Meccanismo ottimo per liste con molti inserimenti ed eliminazioni. Ma è laboriosa la gestione per indice, che richiede iterazione. Anche qui il contains obbliga all'iterazione; non adatto per set e mappe.

Albero binario

Il punto di forza sta nella velocità di ricerca e inserimento, e nel mantenimento dell'ordinamento. Buono per Set e Mappa. (NB: per la mappa bisogna modificare il nodo in modo che contenga anche la chiave).

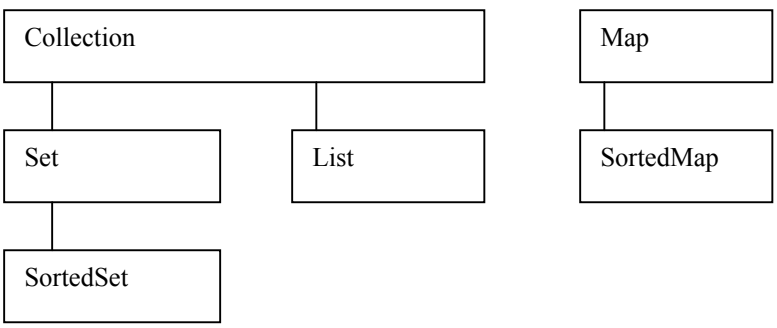
Tavola con Hash

Ha l'oscar della velocità per la ricerca, l'inserimento, la rimozione. Il punto debole è la distruzione di un eventuale ordinamento. E' ideale per le mappe. Ponendo come chiave lo stesso oggetto fornisce una realizzazione ottimale anche del Set.

Meccanismi	*Servizi*		
	Lista	*Set*	*Mappa*
Dynamic Array	***	*	
Catena	**		
Bynary Tree		**	**
Hash		***	***

Le interfacce e classi di java.util

Interfacce



E' definita una interfaccia di base Collection che prevede un servizio ricco. Alcuni metodi sono dichiarati opzionali. Ciò significa che alcune implementazioni possono generare una RuntimeException di "servizio non disponibile".
Esercizio: confrontare le interfacce di java.util con quelle semplificate qui esposte e comprendere le differenze.

Implementazioni di java.util

Classi	Interfacce				
	List	Set	SortedSet	Map	SortedMap
ArrayList	***				
LinkedList	***				
HashMap				***	
HashSet		***			
TreeSet		***	***		
TreeMap				***	***

Esercizio: leggere (e comprendere) il codice di java.util.ArrayList e java.util.HashMap.

Vector e Hashtable

Le due classi Vector e HashTable risalgono alle prime origini di Java, e precedono la sistematizzazione delle collezioni Java. Esse sono rimaste in vita per ragioni di compatibilità con il passato.
Vector è analogo ad ArrayList e HashTable è analoga a HashMap.
Queste due antiche classi hanno la caratteristica di avere tutti i metodi di modifica sincronizzati per motivi di sicurezza in ambienti multi-threading.