Programmazione: concetti, tecniche e linguaggi

Appunti per lo studio di Informatica III A

Angelo Gargantini*

6 novembre 2020

*Alla stesura di questo documento hanno contribuito diversi studenti, vedi le conclusioni

Indice

Introduzione **i**
Teoria della computabilità . . . iii

Programmazione in C **v**
0.1 Dichiarazione . . . v
0.2 Puntatori . . . vi
0.3 Struct . . . viii
0.4 Dichiarazione di sottofunzioni . . . viii

1 Visibilità, funzioni e gestione della memoria **1**
1.1 Blocchi . . . 1
1.2 Premessa . . . 3
1.3 In-line block . . . 4
1.4 Blocchi per cicli e condizioni . . . 5
1.5 Funzioni e procedure . . . 5
1.6 Passaggio di parametri . . . 6
1.6.1 Casi particolari . . . 8
1.7 Variabili globali . . . 8
1.8 Tail recursion . . . 11

2 Type safety **15**
2.1 Tipo . . . 15
2.1.1 Utilizzo . . . 15
2.2 Errori . . . 16
2.2.1 Possibili errori . . . 16
2.3 Sicurezza dei tipi . . . 16
2.4 Type checking . . . 16
2.4.1 Run-time type checking . . . 17
2.4.2 Compile-time type checking . . . 17
2.4.3 Pro e contro . . . 17
2.5 Type Checking in Python . . . 17
2.6 Problemi dei vari linguaggi . . . 18
2.6.1 C/C++ . . . 18
2.6.2 Java . . . 19
2.7 Problemi del C . . . 19
2.7.1 Soluzioni agli errori . . . 22
2.8 Cyclone . . . 25
2.8.1 Materiale, installazione ed esecuzione . . . 25
2.8.2 Situazione unsafe coi puntatori . . . 26
2.8.3 La sintassi di Cyclone . . . 27

3 Programmazione orientata agli oggetti **31**
3.1 Modularità . . . 31
3.2 Astrazione dei dati . . . 31
3.3 Concetti base della programmazione orientata agli oggetti . . . 32
3.3.1 Dynamic lookup . . . 33
3.3.2 Abstraction . . . 33
3.3.3 Subtyping . . . 33
3.3.4 Inheritance . . . 34
3.4 Information hiding in C . . . 34
3.4.1 Primo metodo - uso typedef . . . 35
3.4.2 Secondo metodo uso di varibili di modulo . . . 36
3.4.3 Abstract data type in C . . . 36
3.5 Design Pattern . . . 38
3.5.1 Singleton . . . 38
3.5.2 Facade . . . 39
3.5.3 Visitor . . . 39

4 Java **43**
4.1 Nascita di Java . . . 43
4.2 Obiettivi del linguaggio Java . . . 44
4.2.1 Le decisioni di progettazione . . . 45
4.3 Classi e oggetti/eredità . . . 47
4.4 Pacchetti e Visibilità . . . 50
4.5 Ereditarietà . . . 51
4.6 Classi Astratte e Interfacce . . . 53
4.7 Classificazione dei tipi . . . 55
4.8 Subtyping per classi e interfacce . . . 56
4.9 Array, Covarianza, e Controvarianza . . . 56
4.10 Gerarchia delle classi delle eccezioni . . . 57
4.11 Binding dinamico . . . 60
4.11.1 Overloading dei metodi . . . 60
4.11.2 Overriding, overloading e scelta del metodo da eseguire . . . 61
4.11.3 Il metodo `equals` . . . 66
4.11.4 Covarianza . . . 70
4.11.5 Proprietà delle interfacce . . . 70
4.11.6 Tipi enumerativi . . . 70
4.12 Eccezioni . . . 71
4.13 Varargs . . . 71
4.14 Visibilità in Java . . . 71
4.15 Programmazione generica in Java . . . 72
4.15.1 Metodi generici . . . 75
4.15.2 Bounded type parameters . . . 76
4.15.3 Metodi generici e parametri di tipo delimitati . . . 77
4.15.4 Generics, eredità, e sottotipi (assenza di covarianza dei sottotipi) 77
4.15.5 Classi generiche e sottotipazione . . . 78
4.15.6 Wildcards . . . 78
4.15.7 Upper bounded wildcards . . . 79

4.15.8 Unbounded wildcards . . . 79
4.15.9 Lower bounded wildcards . . . 80

5 C++ 83
5.1 Introduzione . . . 83
5.1.1 Storia . . . 83
5.1.2 Scopo . . . 83
5.2 Classi e dati astratti . . . 84
5.2.1 Astrazione . . . 84
5.2.2 Costruttore . . . 86
5.2.3 Distruttore . . . 88
5.2.4 Funzioni . . . 90
5.2.5 Storage duration e linkage . . . 93
5.2.6 Storage duration . . . 93
5.2.7 Linkage . . . 94
5.3 Incapsulamento . . . 96
5.3.1 Livelli di visibilità . . . 96
5.3.2 Dichiarazioni `friend` . . . 97
5.4 Ereditarietà . . . 97
5.4.1 Costruttori . . . 98
5.4.2 Distruttori . . . 98
5.4.3 Visibilità ed ereditarietà . . . 99
5.4.4 Differenze con Java . . . 100
5.4.5 Ridefinizione . . . 100
5.4.6 Ereditarietà multipla . . . 101
5.5 Polimorfismo . . . 102
5.5.1 Polimorfismo run-time . . . 102
5.5.2 Polimorfismo compile-time: i template . . . 106
5.6 Sottotipazione . . . 109
5.6.1 Slicing . . . 109
5.6.2 Classi astratte . . . 110
5.7 Eccezioni . . . 110
5.8 STL . . . 110
5.8.1 Contenitori . . . 111
5.8.2 Iteratori . . . 114
5.8.3 Algoritmi . . . 116
5.9 Alcune aggiunte non relative agli oggetti . . . 116
5.9.1 Tipo bool . . . 116
5.9.2 Tipi riferimenti . . . 117
5.9.3 Copy constructor e assignment operator . . . 117
5.9.4 Tipo string . . . 117

6 Scala 119
6.1 Introduzione . . . 119
6.1.1 Scala è come Java, ad eccezione di quando non lo è (Y) . . . 119
6.1.2 Scala è conciso . . . 120
6.1.3 Object-oriented, funzionale . . . 121

6.1.4 Dinamicità 122
6.1.5 Altre caratteristiche 123
6.2 Object-orientation in Scala 125
6.2.1 Object 125
6.2.2 Estensione delle classi 126
6.2.3 Classi astratte (esempio) 126
6.2.4 Traits 126
6.3 Function Programming in Scala 127
6.3.1 Operazioni sulle collezioni - Map e Reduce 136

Introduzione

I linguaggi di programmazione sono il mezzo espressivo nell'arte della programmazione dei computer. Un linguaggio di programmazione ideale rende semplice per i programmatori scrivere programmi chiari e funzionanti. Dato che i programmi sono ideati per essere compresi, modificati e mantenuti lungo il corso del loro ciclo di vita, un corretto stile di programmazione permette a chiunque legga i programmi di comprendere come si comportano.

Un buon linguaggio per una larga scala di programmi sarà in grado di aiutare i programmatori a gestire le interazioni tra le componenti software efficacemente. Valutando un linguaggio di programmazione dobbiamo considerare i processi di progettazione, implementazione, testing e mantenimento del software, chiedendoci come ogni linguaggio supporti questi passaggi del ciclo di vita di un software.

Ci sono differenti trade-off nella progettazione di un linguaggio di programmazione. Alcuni linguaggi rendono semplice la stesura veloce di programmi, ma possono rendere difficile la creazione di strumenti adibiti al testing. Certi costrutti di un linguaggio rendono più semplice per un compilatore l'ottimizzazione del programma finale, ma possono renderne la programmazione ingombrante. Dato che differenti ambienti di sviluppo richiedono differenti caratteristiche di programmazione, esistono differenti linguaggi di programmazione che, orientati verso specifici obiettivi, hanno applicato scelte differenti in merito ai trade-off imposti dalla progettazione di un linguaggio di programmazione.

Nonostante ciò i linguaggi di programmazione possiedono alcuni concetti comuni:

- variabile;
- istruzione;
- espressione;
- struttura di controllo;
- sottoprogramma;
- struttura dati;
- sintassi;
- semantica di esecuzione;
- tipo;

e altri concetti non in comune:

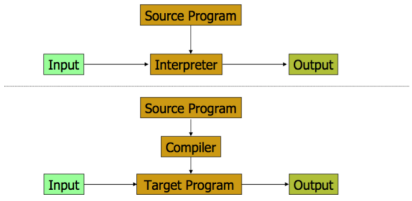
- analisi (statica o dinamica);
- rapporto tra espressività ed efficienza;

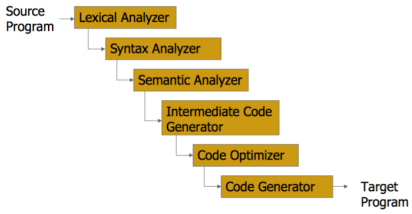
- categoria di appartenenza.

Consideriamo ora l'analisi di un programma scritto in un determinato linguaggio; essa può avvenire principalmente in due modi: o staticamente prima della sua esecuzione o nel corso di quest'ultima.

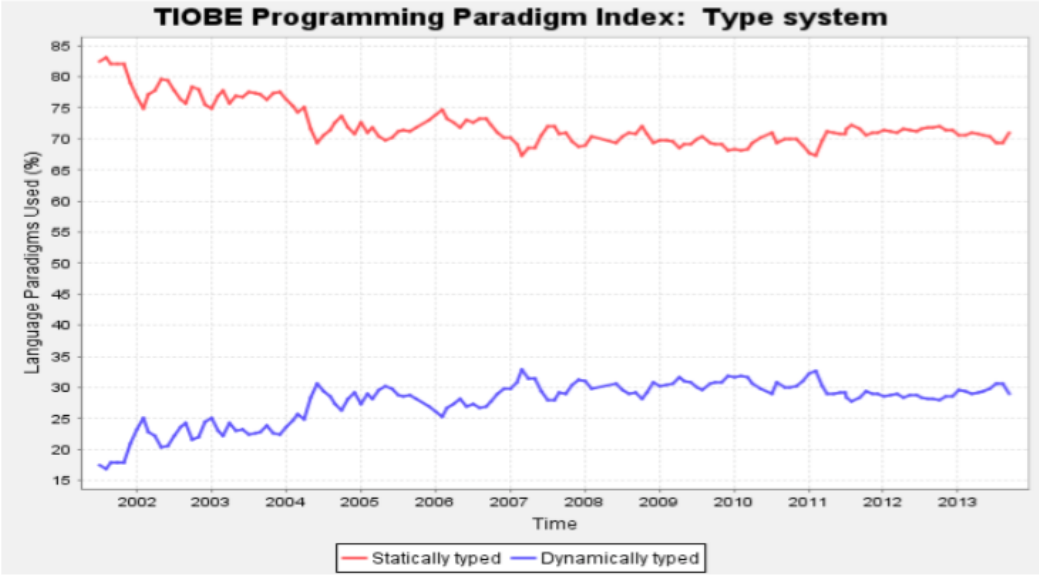
Nel primo caso si parla di linguaggi di programmazione compilati, dato che prima della loro esecuzione subiscono un processo di controllo, chiamato appunto compilazione, in grado di verificare la correttezza del codice prodotto. Questo tipo di analisi permette di assicurarsi che un codice, una volta compilato, sia "relativamente corretto" (in realtà scopriremo che può essere soggetto anche ad altri tipi di errori).

Nel secondo caso il codice prodotto viene controllato istruzione per istruzione direttamente durante l'esecuzione. Ciò permette la corretta esecuzione anche di programmi solo parzialmente esatti; ad esempio, nel caso in cui durante l'esecuzione non vengano raggiunte istruzioni sbagliate, il programma tenderà ad ignorarle rischiando di illudere il programmatore che il programma da lui scritto sia corretto.





Statically Typed Languages	71.0%
Dynamically Typed Languages	29,0%



Un'altra peculiarità distintiva per un linguaggio di programmazione è la categoria cui appartiene. In particolare le principali categorie che classificano un linguaggio di

Tabella 1: Diffusione delle varie categorie di linguaggi.

Categoria	Settembre 2012 [%]	Variazione annua [%]
Object-oriented	56.0	−1.1
Procedurale	37.3	−0.9
Funzionale	3.8	+0.6
Logico	3.0	+1.3

programmazione sono:

- linguaggi procedurali (Pascal, C, ...);
- linguaggi object-oriented (Java, C#, ...);
- linguaggi funzionali (Lisp, ...);
- linguaggi logici (Prolog, ...).

Prima di concludere questo capitolo è bene fare un piccolo approfondimento sulla differenza tra sintassi e semantica: la prima identifica i simboli utilizzati per la stesura di un programma, mentre la seconda le azioni che si verificano quando quest'ultimo viene eseguito. Ciò significa che un linguaggio di programmazione si occupa di tramutare la sintassi dei simboli con cui viene scritto nella semantica delle azioni che devono essere compiute per raggiungere gli obiettivi che il programmatore si è prefissato.

Dunque, dato che la correttezza vera e propria di un programma non corrisponde alla sua correttezza sintattica, un compilatore in grado di controllare solamente quest'ultimo tipo di correttezza non potrà garantire il definitivo buon funzionamento del programma. Per questo motivo per la validazione finale del programma è opportuno definire un'approfondita fase di test volta a verificarne l'affidabilità attraverso diverse esecuzioni.

Teoria della computabilità

La teoria della computabilità stabilisce alcune importanti regole generali per la progettazione e lo sviluppo di linguaggi di programmazione. Di seguito si possono trovare i concetti fondamentali da ricordare riguardo questo argomento:

Parzialità Funzioni definite ricorsivamente possono essere funzioni parziali. Cioè, possono non essere sempre funzioni totali ma svolgere solo alcune parti elementari della procedura richiesta.

Computabilità Alcune funzioni sono computabili e altre no. I linguaggi di programmazione permettono di definire funzioni computabili; non possiamo scrivere programmi per funzioni che non sono computabili in principio.

Completezza di Turing Tutti i linguaggi di programmazione multiuso standard offrono la stessa classe di funzioni computabili.

Indecidibilità Molte importanti proprietà dei programmi non possono essere determinate da una funzione computabile.

Programmazione in C

Per questo corso, la programmazione in C è data per assunta. Ripassiamo un po' il suo uso.

0.1 Dichiarazione

Librerie / funzioni

```
#include <stdio.h>    // Direttive di inclusione
#include <stdlib.h>   // per il pre-processore

int main() {          // dichiarazione di funzione
  printf("Hello World!\n");
  return 0;
}
```

Variabili

Per *dichiarare* una variabile si scrive:

```
Tipo nome;                  // Es.: float media;
Tipo nome1, nome2;          // Es.: int i, count, somma;
Tipo nome1, nome2, nome3;
```

Le *variabili globali* si definiscono al di sopra della funzione `main()`:

```
short number;
char letter;
int main() { return 0; }
```

E' *possibile inizializzare* una variabile utilizzando l'operatore di assegnazione "=":

```
int sum = 0, number;
```

typedef

Si possono definire *nuovi tipi* di variabili utilizzando l'istruzione `typedef` (questo risulta utile soprattutto quando si creano strutture complesse di dati).

Dichiaro due tipi di variabili:

```
typedef float real;    // Ho creato il tipo 'real'
typedef char letter;   // Ho creato il tipo 'letter'
```

Ora dichiaro due variabili che hanno come tipo quelli appena creati:

```
real sum = 0.0;
letter nextLetter;
```

0.2 Puntatori

Definizione

Un puntatore è un tipo di dato e definisce una variabile che contiene l'indirizzo in memoria di un'altra variabile. Si possono avere puntatori a qualsiasi tipo di variabile.

- La dichiarazione di un puntatore include il tipo dell'oggetto a cui il puntatore punta.
- L'operatore `&` fornisce l'indirizzo di una variabile.
- L'operatore `*` dà il contenuto dell'oggetto a cui punta un puntatore.

Per dichiarare un puntatore ad una variabile l'istruzione è la seguente:

```
TipoPuntato *nome_puntatore;  // Es.: int *pointer;
```

Esempi:

```
int *ptr;           // Dichiara ptr come un puntatore a int
int x = 1, y = 2;
ptr = &x;           // Assegna a pointer l'indirizzo di x
y = *ptr;           // Assegna a y il contenuto di ptr
x = ptr;            // Assegna ad x l'indirizzo contenuto in
   ptr
*ptr = 3;           // Assegna al contenuto di ptr il valore 3
```

Array e puntatori

Un array in C è un blocco di dati messi in sequenza. Si dichiarano usando la seguente sintassi. Si possono inizializzare quando si dichiarano.

```
dataType arrayName[arraySize];
..
int mark[5] = {19, 10, 8, 17, 9};
```

Posso accedere agli elementi usando [] e all'indirizzo degli elementi usando & e [].

```
int a[10];
printf("Address of array x: %p", &a[0]);
printf("Address of array x: %p", &a[1]);
```

Nella maggior parte dei contesti, i nomi degli array *decadono* in puntatori. In parole semplici, i nomi degli array vengono convertiti in puntatori alla prima cella dell'array. Questo è il motivo per cui puoi usare i puntatori per accedere agli elementi degli array. Tuttavia, dovresti ricordare che i puntatori e gli array non sono la stessa cosa. Ad esempio, non puoi cambiare il valore di un nome dell'array.

```
int a[10];
*a = 10; // equivalent to a[0] = 10
int x;
a = &x;/ ERRORE!!!
```

Aritmetica dei puntatori

Un array di elementi può essere pensato come disposto in un insieme di locazioni di memoria consecutive:

```
int a[10], x;
int *ptr;
ptr = &a[0];    // ptr punta all'indirizzo di a[0]
x = *ptr;       // x = contenuto di ptr (in questo caso, a[0])
```

Ora potremo incrementare ptr con successive istruzioni

```
++ptr
```

Potremo anche avere `(ptr + i)` che è equivalente ad `a[i]`, con $i = 0, 1, 2, \ldots, 9$.

Nota che l'aritemica dei puntatori tiene conto della dimensione del tipo delle celle puntate e di quanti byte di memoria occupano, quindi se faccio `p++` con un puntatore, il vero incremento in termini di byte dipenderà dal tipo puntato da `p`, se è ad esempio `int` o `double` avrà effetto diverso.

Allocazione dinamica della memoria: `malloc/free`

La funzione `malloc` alloca nuova memoria e la restituisce:

```
void *malloc(int number_of_bytes)
```

L'uso di `void*` indica puntatore generico. Per questo alcune volte è necessario fare il cast del puntatore. Spesso viene utilizzata assieme alla funzione `sizeof()`:

```
int *ip = (int*) malloc(100 * sizeof(int));
```

L'uoperatore unario `sizeof` restituisce il numero di byte occupati dall'argomento che può essere un tipo (es. `sizeof(int)`) o anche una espressione (es. `sizeof(x)`).

La memoria allocata tramite malloc è *persistente*: ciò significa che continuerà ad esistere fino alla fine del programma o fino a quando non sarà esplicitamente *deallocata* (liberata) dal programmatore. Questo risultato è ottenuto tramite la funzione *free*

```
void free (void *pointer)  // Es.: free(cp);
```

Alcuni linguaggi, come per esempio Java, non hanno l'istruzione `free` ed usano invece il *garbage collector*.

Stringhe, inizializzazione, confronto e manipolazione Le stringhe sono in realtà un array unidimensionale di caratteri terminati da un carattere nullo `'\0'` (come intero vale 0). Così una stringa null-terminata contiene i caratteri che compongono la stringa seguita da un null.

La seguente dichiarazione e inizializzazione creano una stringa costituita dalla parola "Ciao". Per contenere il carattere nullo alla fine dell'array, la dimensione dell'array di caratteri contenente la stringa è più che il numero di caratteri nella parola "Ciao".

```
char saluto [5] = {'C', 'i', 'a', 'o', '\ 0'};
```

Se si segue la regola dell'inizializzazione dell'array allora è possibile scrivere l'istruzione di cui sopra come segue -

```
char saluto [] = "Ciao";
```

Alcune funzioni che si usano per le stringhe:

```
strcpy(d, s);
```

Copia la stringa *s* nella stringa *d*. Attenti alle lunghezze dei buffer.

```
strcat(s1, s2);
```

Concatena la stringa s2 alla fine della stringa s1.

```
strlen(s1);
```

Restituisce la lunghezza di una stringa (zero-terminated)

```
strcmp(s1, s2);
```

Returns 0 if s1 and s2 are the same; less than 0 if s1<s2; greater than 0 if s1>s2.

0.3 Struct

```
struct Book {
   char  title[50];
   char  author[50];
   char  subject[100];
   int   book_id;
}; // attenti al ;

int main( ) {

   struct Book Book1;        /* Declare Book1 of type Book */
   struct Book Book2;        /* Declare Book2 of type Book */

   /* book 1 specification */
   strcpy( Book1.title, "C Programming");
   /* ... */
}
```

Si può usare typedef se si vuole evitare di mettere struct ogni volta:

```
typedef struct B {
        char  title[50];
} Book;

Book b1;
```

0.4 Dichiarazione di sottofunzioni

La forma generale di una definizione di funzione nel linguaggio di programmazione C è la seguente:

```
return_type function_name (parameter list) {
        body of the function
}
```

La definizione di funzione nella programmazione C consiste in un'intestazione di funzione e un corpo di una funzione. Ecco tutte le parti di una funzione:

return_type : una funzione può restituire un valore. Return_type è il tipo di dati del valore restituito dalla funzione. Alcune funzioni eseguono le operazioni desiderate senza restituire un valore. In questo caso, return_type è la parola chiave void.

function_name : è il nome effettivo della funzione. Il nome della funzione e l'elenco dei parametri insieme costituiscono la firma (signature) della funzione.

parameter list : un parametro è come un segnaposto. Quando viene invocata una funzione, si passa un valore al parametro. Questo valore viene definito parametro o argomento effettivo. L'elenco dei parametri si riferisce al tipo, all'ordine e al numero dei parametri di una funzione. I parametri sono opzionali; cioè, una funzione potrebbe non contenere parametri.

body : Il corpo della funzione contiene una serie di istruzioni che definiscono cosa fa la funzione.

Un esempio:

```
/* function returning the max between two numbers */
int max(int num1, int num2) {
        /* local variable declaration */
        int result;
        if (num1 > num2)
                result = num1;
        else
                result = num2;
        return result;
}
```

1 Visibilità, funzioni e gestione della memoria

Le caratteristiche di un linguaggio di programmazione che permette l'associazione tra le parole all'interno del programma e le locazioni di memoria interessate sono due:

Scope Permette a due parole sintatticamente identiche di riferirsi a differenti locazioni di memoria.

Function call (chiamata di funzione) Richiede una nuova area di memoria dove salvare i parametri della funzione e le variabili locali.

Assumiamo che il linguaggio di programmazione abbia le seguente caratteristiche:

- Non sia necessario che il programmatore dichiari tutte le variabili che vorrà usare fin dall'inizio: alcune variabili potranno essere necessarie solo in alcune zone di programmi e poi potranno essere "dimenticate".
- Il programma potrà essere diviso in sotto programmi ognuno dei quali avrà alcuni dati di in ingresso, altri di uscita e potrà avere dei dati locali.
- Sarà possibile ??

1.1 Blocchi

Molti moderni linguaggi di programmazione prevedono l'utilizzo di *blocchi*. Un blocco è una regione del programma scritto, identificata da un carattere che ne individua l'inizio e la fine, che può contenere dichiarazioni locali riferite a questa regione. Di seguito alcune righe di codice C che illustrano meglio il concetto appena spiegato:

```
{ // outer block
  int x = 0;
  int y = x + 1;
  { // inner block
    int z = (x + y) * (x - y);
  }
}
```

Nell'esempio riportato le variabili `x` e `y` sono dichiarate nel blocco più esterno (*outer block*) e la variabile `z` in quello più interno (*inner block*). Una variabile dichiarata all'interno di in un blocco che ne include altri è ritenuta globale rispetto ai blocchi interni. Nel nostro caso `x` e `y` sono locale all'*outer block* ma globale rispetto all'*inner block*, mentre `z` è solamente locale all'*inner block*.

L'utilizzo dei blocchi è una premessa fondamentale all'utilizzo della ricorsione nei linguaggi di programmazione.

I linguaggi costituiti da una struttura a blocchi sono caratterizzati dalle seguenti proprietà:

- Nuove variabili possono essere dichiarate in vari punti del programma.
- Ogni dichiarazione è visibile in una certa regione del programma chiamata blocco. I blocchi possono essere annidati ma non possono essere parzialmente sovrapposti. In altre parole, se due blocchi contengono delle espressioni o istruzioni in comune significa che uno dei due deve essere interamente contenuto nell'altro.
- Quando un programma inizia ad eseguire le istruzioni contenute in un blocco a runtime, la memoria è allocata per le variabili dichiarate in quel blocco.
- Quando un programma esce da un blocco, la memoria allocata alle variabili dichiarate in quel blocco sarà almeno parzialmente deallocata.
- Un identificatore che non è dichiarato nel blocco corrente è considerato globale e si riferisce all'entità con quel nome che è dichiarata nel blocco più vicino tra quelli che includono il blocco corrente.

Prima di proseguire con l'analisi dei meccanismi di gestione della memoria è bene richiamare il significato di alcuni termini:

Stack Zona di memoria gestita secondo i principi di inserimento ed estrazione di una pila.

Record di attivazione Struttura dati utilizzata per l'organizzazione dei dati associati ad una particolare regione del programma.

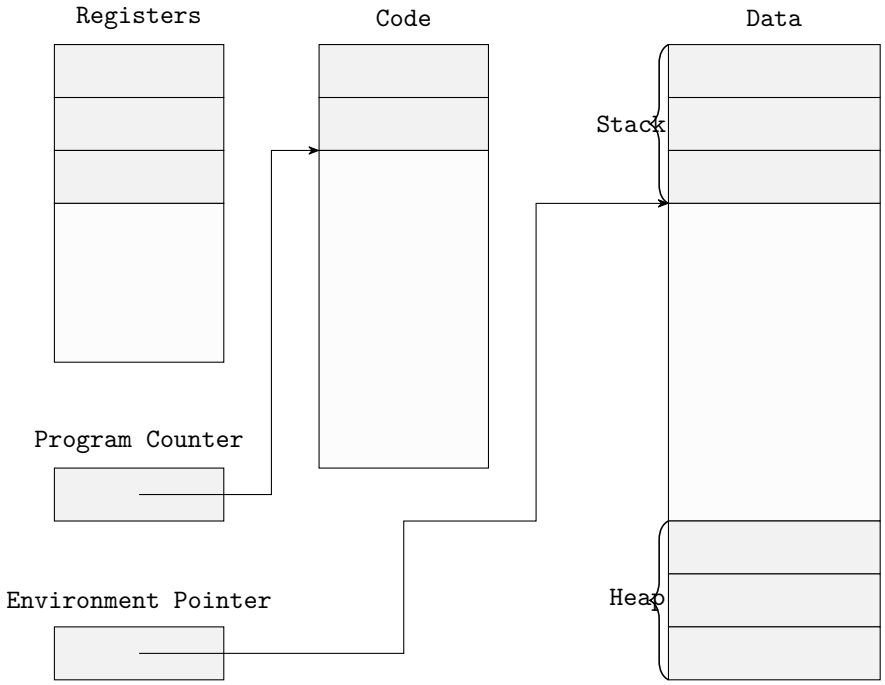
Consideriamo ora tre classi di variabili per descrivere i meccanismi di gestione della memoria:

Variabili locali Sono salvate nello stack del record di attivazione associato al blocco.

Parametri (di funzioni o blocchi di procedure) Sono salvati nei record di attivazione associati ai blocchi.

Variabili globali Sono dichiarate in uno dei blocchi più esterni e a cui si può accedere attraverso un record di attivazione posizionato sullo stack di runtime prima dell'attivazione del blocco corrente.

1.2 Premessa



Prima di inoltrarci oltre nella spiegazione di come i più comuni linguaggi di programmazione sviluppano la gestione della memoria di un programma, è bene premettere qualche indicazione sul modello di rappresentazione utilizzato come riferimento.

Nella figura riportata qui sopra è rappresentata il modello semplificato di come i linguaggi strutturati a blocchi gestiscono la memoria, secondo il quale la memoria dedicata al codice viene divisa dalla memoria adibita ai dati.

Il *program counter* è un particolare registro che si occupa di salvare l'indirizzo dell'istruzione corrente. Normalmente questo registro viene incrementato ad ogni ciclo, in modo da puntare ordinatamente a tutte le istruzioni del programma. Tuttavia, quando il programma entra in un nuovo blocco, un record di attivazione contenente lo spazio per le variabili locali dichiarate nel blocco viene aggiunto al run-time stack (disegnato nel nostro caso in cima alla memoria dati); a questo punto l'environmentpointer viene impostato perché punti al record di attivazione appena creato. Quando il programma esce dal blocco, il record di attivazione viene rimosso dallo stack e l'environmentpointer ritorna a puntare alla sua precedente locazione.

Il programma può, inoltre, salvare i dati all'interno dello heap, in modo che sopravvivano al termine del blocco corrente.

La regola per cui il record di attivazione allocato più recentemente è anche il primo ad essere deallocato viene definita come *disciplina di stack*.

La maggior parte dei linguaggi con struttura a blocchi sono implementati tramite uno stack e l'utilizzo di *higher-order functions* può causare malfunzionamenti con la disciplina di questa particolare struttura dati.

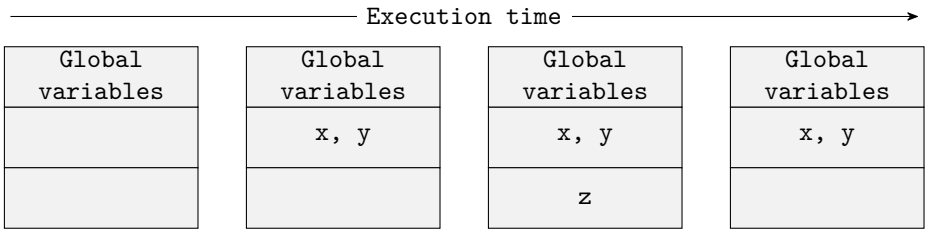
Un *in-line block* è un blocco che non funge né da corpo di una funzione né di una procedura.

La differenza tra *Scope* e *Lifetime* (tempo di vita) di una locazione è la seguente: lo *Scope* è la regione di testo all'interno della quale la dichiarazione è visibile, il *Lifetime* è la durata (riferita ad un esecuzione del programma) in cui una locazione è allocata come risultato di una dichiarazione. Infatti, nel caso in cui in un blocco interno ad un altro dichiarassimo una variabile locale con nome uguale a quello di una variabile dichiarata nel blocco più esterno, la prima oscurerebbe la seconda durante tutto il suo *Scope*. In questo caso il *Lifetime* della variabile più interna risulterebbe contenuto in quello della variabile più esterna, ma lo stesso discorso non potrebbe valere, invece, per il suo *Scope*.

1.3 In-line block

Quando un programma entra in un *in-line block* alloca subito un record di attivazione adibito a contenere le variabili locali qui dichiarate sullo stack. All'uscita del blocco questo record di attivazione verrà poi deallocato, liberando la parte di stack che occupava.

```
{ // outer block
  int x = 0;
  int y = x+1;
  { // inner block
    int z = (x+y)*(x-y);
  }
}
```

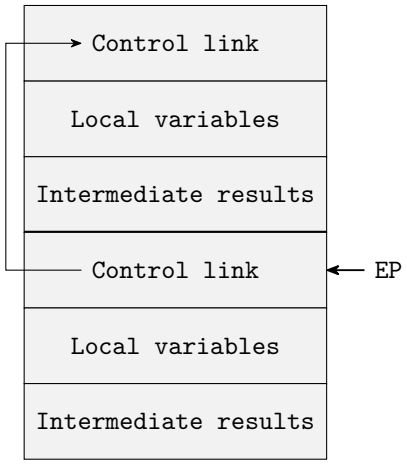


Alcuni compilatori, soprattutto in passato, prevedevano all'interno del record di attivazione anche dello spazio per l'allocazione dei risultati intermedi di istruzioni lunghe. Tuttavia nei moderni calcolatori ci sono registri a sufficienza per evitare questa ulteriore allocazione, ottimizzando la gestione di memoria.

Dato che i record di attivazione di blocchi differenti possono avere dimensioni diverse ognuno di questi record contiene un campo, chiamato *control link* (o *dynamic link*), che punta alla cima del record di attivazione precedente.

Dunque, ogni volta che si accede ad un nuovo blocco, il *control link* all'interno del nuovo record di attivazione viene impostato in modo che contenga l'attuale valore dell'*environment pointer*. Quest'ultimo, invece, verrà modificato in modo che punti al nuovo record di attivazione.

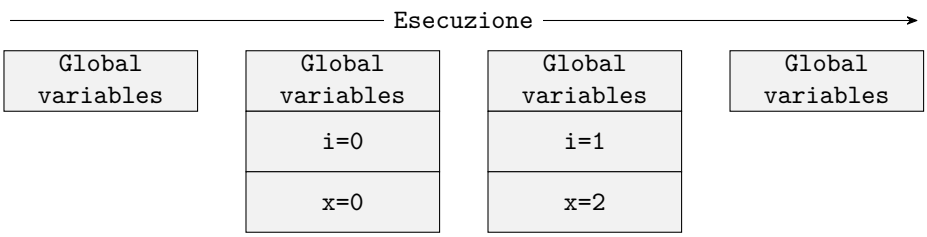
Così facendo il *control link* conterrà il valore da reinserire all'interno dell'*environment pointer* alla fine di ogni blocco, cosicché il programma possa tornare a lavorare sul precedente record di attivazione (e cioè quello del blocco più esterno a quello da cui si sta uscendo).



1.4 Blocchi per cicli e condizioni

Del tutto simili sono il trattamento dei blocchi in corrispondenza a condizioni o cicli (`for`, `while`, etc...). Se nel ciclo è dichiarata una variabile, all'entrata del ciclo viene riservato lo spazio che viene poi liberato all'uscita del ciclo (NOTA: non viene aggiunto un blocco ad ogni ciclo). Anche le variabili del ciclo for, se sono dichiarate nel for, sono create all'entrata del ciclo e poi deallocate quando il ciclo finisce. Ad esempio:

```
for(int i = 0; i < 2; i++){
 int x = i*2;
}
```



1.5 Funzioni e procedure

In generale esiste una distinzione formale tra questi due termini. Infatti, per definizione, una funzione ritorna sempre un valore al programma chiamante mentre una procedura non ha parametri di ritorno. Tuttavia, non è obiettivo di questa dispensa fornire una definizione di questi termini o mantenere questi due concetti distinti, quindi, verranno trattati come sinonimi per il resto degli appunti.

Dato che una funzione può essere richiamata in diverse parti del programma, ogni record di attivazione associato alla chiamata di una funzione deve contenere i seguenti campi:

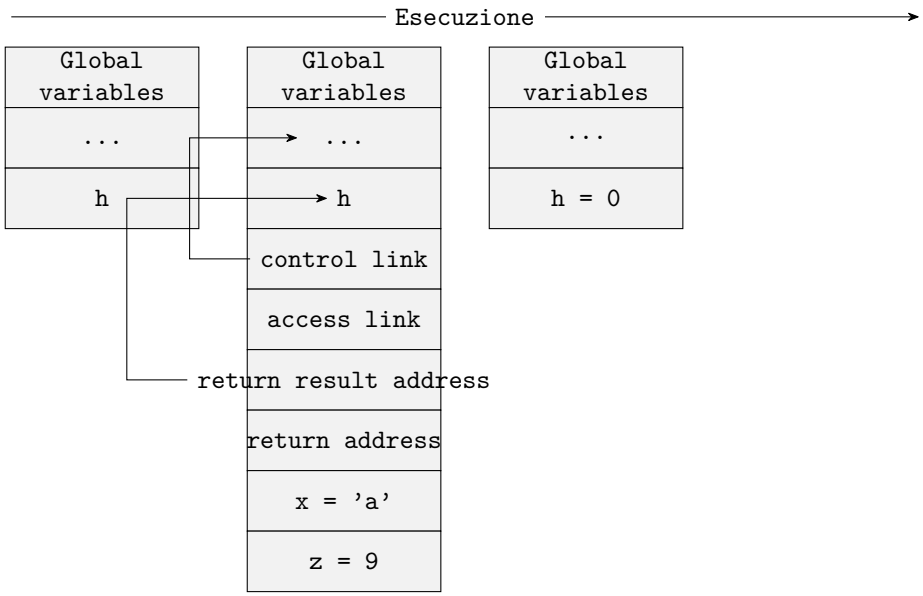
- *control link*, che punta al record di attivazione precedente sullo stack;
- *access link*, di cui discuteremo più avanti;

- *return address*, contenente l'indirizzo della prima istruzione da eseguire quando la funzione termina;
- *return-result address*, contenente l'indirizzo dove salvare il risultato della funzione;
- parametri della funzione;
- variabili locali dichiarate all'interno della funzione;
- spazio adibito all'elaborazione di risultati intermedi.

Queste informazioni possono essere salvate in ordine e modo differenti a seconda dell'implementazione linguaggio. Questo modello è illlustrato puramente a scopo didattico quindi non terrà conto di eventuali approssimazioni che possono essere compiute da alcuni compilatori. Inoltre, noi per comodità abbiamo riportato i nomi delle variabili nel record di attivazione, mentre nella realtà vengono tutte sostituite da indirizzi di memoria in fase di compilazione.

Ad esempio:

```
int foo(char x){
        int z = 9;
        return 0;
}
...
int h = foo('a');
```



1.6 Passaggio di parametri

Il passaggio di parametri ad una funzione può avvenire in moltissimi modi a seconda del linguaggio di programmazione. Però concettualmente possiamo considerare due modi principali:

- per valore (un R-value, il contenuto di una locazione o una espressione)
- per riferimento (non il contenuto ma la locazione/contenitore stesso, un L-value)

Prendiamo come riferimento il C++ anche se simili passaggi esistono anche in altri linguaggi di programmazione.

In caso di passaggio per **valore** l'espressione che genera il parametro viene risolta prima chi inizi la funzione ed il suo risultato salvato direttamente sullo stack; se invece si utilizza una specifica variabile per passare un parametro durante la chiamata di una funzione, quest'ultima non sarà comunque in grado di modificare il contenuto della variabile utilizzata dal programma chiamante.

In caso di passaggio per **riferimento**, invece, non è possibile utilizzare un'espressione per il passaggio parametri, ciò perchè sul record di attivazione della funzione che si vuole chiamare non verrà salvato il valore della variabile usata per il passaggio, bensì, il suo indirizzo; ciò significa che ogni modifica alla variabile che avverrà durante l'esecuzione della funzione si ripercuoterà inevitabilmente sulla variabile utilizzata dal programma chiamante per il passaggio del parametro.

Seguiamo i seguenti esempi per comprendere meglio questo concetto:

```
void f(int x, int y, int z) {
  x = y + z;
}

int main() {
  int a = 10, b = 30;
  f(a, 2*a, b);        // Corretto
  printf("%d\n", a);   // Stampa 10
  return 0;
}
```

Indirizzo	Variabile	Valore
200	x	10
201	y	20
202	z	30

Il primo esempio riporta del codice che effettua passaggio di parametri per valore. Qui è possibile passare parametri utilizzando un'espressione ed il risultato (o il contenuto della variabile utilizzata) viene registrato direttamente all'interno del record di attivazione. Se la variabile usata per il passaggio viene modificata durante il corso della funzione queste modifiche non vengono riportate nel programma chiamante dato che agiscono solamente su una variabile locale, interna alla procedura.

```
void f(int &x, int &y, int &z) {
  x = y + z;
}

int main() {
  int a = 10, b = 30;
//f(a, 2*a, b);        // Sbagliato
  f(a, b, b);          // Corretto
```

```
  printf("%d\n", a);  // Stampa 60
  return 0;
}
```

Indirizzo	Variabile	Valore
150	a	10
151	b	30
...	...	...
200	x	150
201	y	151
202	z	151

Il secondo esempio, invece, riporta del codice che effettua passaggio di parametri per riferimento. In questo caso, infatti, non è possibile passare parametri utilizzando un'espressione, quindi la chiamata della funzione è stata modificata (nei commenti è rimasto il codice originario). Come possiamo notare dalla tabella rappresentante il record di attivazione, nella cella allocata ai parametri `x`, `y` e `z` sono allocati gli indirizzi delle variabili dichiarate nel programma chiamante. Ovviamente, in questo caso ogni modifica sulle variabili durante il corpo della funzione si ripercuote direttamente sulle variabili usate per il passaggio.

Compreso il funzionamento di queste tipologie di passaggio, possiamo quindi intuire che mentre il passaggio per valore è molto più sicuro, in quanto isola gli effetti delle operazioni, il passaggio per riferimento offre molte più possibilità al programmatore e permette di risparmiare spazio di memoria in occasione di strutture dati di grandi dimensioni, dato che si limita a salvare solamente un indirizzo per ogni parametro senza copiarne l'intero contenuto.

1.6.1 Casi particolari

Vedi sulle **slide**.

Passaggio mediante puntatore

Il passaggio di puntatori è un passaggio per valore, ma si usa (in C) per ottenere lo stesso effetto del passaggio per riferimento.

Passaggio di array

Passaggio di stringhe

Passaggio di struct

Passaggio di puntatore a puntatore

Passaggio di parametri in Java

1.7 Variabili globali

Prima di descrivere come vengono trattate le variabili globali nei record di attivazione è bene fare una piccola digressione specificando la differenza tra *static scope* e *dynamic*

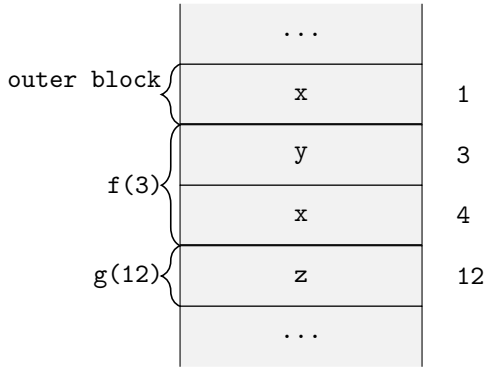
scope.

Nel primo caso l'identificatore globale fa riferimento alla variabile con lo stesso nome dichiarata nel blocco più vicino tra quelli che contengono quello in esecuzione. Nel secondo caso, invece, fa riferimento alla variabile con lo stesso nome dichiarata nel record di attivazione più vicino.

Queste due definizioni possono sembrare molto simili eppure creano una sostanziale differenza nell'approccio scelto da un particolare linguaggio di programmazione. Infatti, mentre i principali linguaggi di programmazione utilizzano uno static scope, alcuni linguaggi con particolari esigenze ed ambiti di lavoro utilizzano il dynamic scope.

Per meglio comprendere la differenza tra questi due concetti ci avvaliamo, ancora una volta, di un esempio pratico:

```
int x = 1;
int g(int z) {
  return x + z;
}
int f(int y) {
  int x = y + 1;
  return g(y * x);
}
f(3);
```

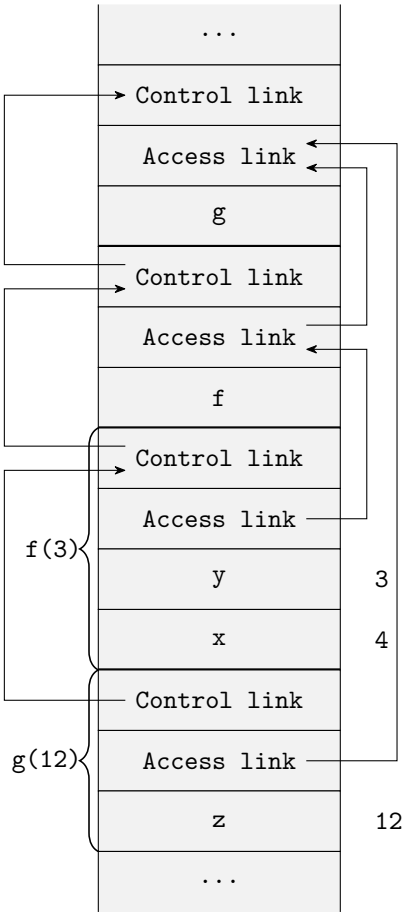


Come possiamo notare ci sono due salvataggi di `x` sul nostro stack. Il dynamic scope ci porterebbe a considerare la locazione contenente 4, essendo il record di attivazione più vicino all'interno dello stack a quello in esecuzione al momento. Tuttavia, osservando il codice, ci accorgiamo che lo static scope ci porterebbe invece a considerare la locazione contenente 1, in quanto dichiarata nel blocco di codice contenente la funzione `g`. Dunque questo esempio dimostra chiaramente come un diverso approccio nell'implementazione di un linguaggio di programmazione può portare a risultati differenti nell'interpretazione del medesimo codice.

Riprendiamo ora il campo *access link* precedentemente nominato nella descrizione di un record di attivazione. Questo campo, all'inizio del record di una funzione, contiene l'indirizzo del blocco più esterno, a livello di codice, riferito alla funzione e serve per imporre sempre un approccio static scope.

Ovviamente per gli in-line block questo campo si fa superfluo, dato che in ogni caso il blocco più esterno che dovrebbe essere puntato coinciderebbe proprio con il "blocco chiamante", e quindi il record di attivazione più vicino sullo stack. Invece, nelle funzioni il blocco più esterno alla dichiarazione nel codice raramente coincide con il blocco chiamante, e cioè il record di attivazione più vicino sullo stack.

Avvaliamoci ancora di un esempio per comprendere meglio il funzionamento degli access link; l'immagine seguente rappresenta i record di attivazione presenti sullo stack riferiti al codice precedente, descritti in maniera più dettagliata:



Per comodità e chiarezza di rappresentazione supponiamo che ogni dichiarazione di funzione occupi un record di attivazione a sé stante. Come già visto i control link puntano sempre al record di attivazione chiamante, e cioè quello appena precedente sullo stack. Invece, gli access link puntano di volta in volta al record di attivazione del blocco più esterno all'interno del codice.

Infatti, nel nostro caso l'access link della chiamata g(12), la quale si riferisce ovviamente alla funzione g, punta proprio allo stesso campo all'interno del record di attivazione che descrive la dichiarazione dell'omonima funzione (il primo dall'alto in figura). Viceversa, il medesimo campo della chiamata f(3) punta al campo corrispondente interno al record di attivazione della funzione f. Questa regola procede allo stesso modo di blocco in blocco fino a raggiungere quello più esterno che conterrà le variabili globali del programma. Infatti, si può supporre che più in alto del primo record di attivazione rappresentato in figura ci sia quello più esterno possibile del programma, contenente la dichiarazione della variabile x.

Riassumendo, il control link punta sempre all'ultimo blocco che è stato chiamato, mentre l'access link punta al blocco che contiene quello attuale all'interno del codice.

Dunque, il control link dipende dal comportamento dinamico del programma, mentre l'access link da quello statico.

L'utilità degli access link si manifesta solamente nei linguaggi di programmazione in cui è prevista la dichiarazione di funzioni all'interno di altre, negli altri, come ad esempio C, questi campi possono risultare superflui.

1.8 Tail recursion

In questa sottosezione esaminiamo un'utile ottimizzazione del compilatore chiamata eliminazione della ricorsione della coda o *tail recursion*. Per le funzioni ricorsive di coda, che verranno successivamente descritte, è possibile riutilizzare un record di attivazione per una chiamata ricorsiva alla funzione. Ciò riduce la quantità di spazio utilizzato da una funzione ricorsiva. Il principale concetto di linguaggio di programmazione di cui abbiamo bisogno è il concetto di *tail call* (chiamata in coda). Supponiamo che la funzione f chiama la funzione g. Le funzioni f e g potrebbero essere funzioni diverse o potrebbero essere la stessa funzione. Una chiamata a f nel corpo di g è una tail call se g restituisce il risultato della chiamata a f senza ulteriori calcoli. Ad esempio, nella funzione (in pseudo codice)

```
fun g(x): if x = 0 then f(x) else f(x)*2
```

la prima chiamata a f nel corpo di g è una tail call, poiché il valore di ritorno di g è esattamente il valore di ritorno della chiamata a f. La seconda chiamata a f nel corpo di g non è una tail call perché g esegue un calcolo che coinvolge il valore di ritorno di f prima che g ritorni. Una funzione f è tail recursive (ricorsiva in coda) se tutte le chiamate ricorsive nel corpo di f sono tail call a f.

Ecco una funzione che calcola il fattoriale utilizzando solo ricorsione in coda:

```
fun tlfact(n, a): if n <= 1 then a else tlfact(n-1, n*a)
```

Più specificamente, per qualsiasi numero intero positivo n, `tlfact (n, 1)` restituisce n!. Possiamo vedere che `tlfact` è una funzione ricorsiva in coda perché l'unica chiamata ricorsiva nel corpo di tlfact è una tail call. Il vantaggio della ricorsione della coda è che possiamo usare lo stesso record di attivazione per tutte le chiamate ricorsive. Considera la chiamata `tlfact(3,1)`. La figura 1.1 mostra le parti di ciascun record di attivazione nel calcolo che sono rilevanti per la discussione.

Al termine della terza chiamata, passa il risultato di ritorno alla seconda chiamata, che quindi restituisce il risultato di ritorno alla prima chiamata. Possiamo semplificare questo processo lasciando che la terza chiamata restituisca il suo risultato all'attivazione che ha effettuato la chiamata originale, `tlfact (3,1)`. In questo scenario, quando la terza chiamata termina, possiamo deallocare anche il record di attivazione per la seconda chiamata, poiché non ne avremo più bisogno. Lo stesso tra la prima e la seconda chiamata: poiché il record di attivazione per la prima chiamata non è più necessario all'inizio della seconda chiamata, è possibile deallocare il record di attivazione della prima chiamata. Ancora meglio, invece di deallocare il primo e quindi allocare un secondo record di attivazione identico al primo, possiamo usare un unico record di attivazione per tutte e tre le chiamate. La figura 1.2 mostra come lo stesso record di attivazione può essere usato tre volte per tre chiamate successive per adeguare il risultato ricorsivo. La figura mostra il contenuto di questo record di attivazione per ogni chiamata. Quando inizia la prima chiamata, tlfact (3,1), viene allocato un record di attivazione con parametri (1,3). Quando inizia la seconda chiamata, tlfact (2,3), i valori dei parametri vengono cambiati in (2,3). L'eliminazione della tail recursion riutilizza un singolo record di attivazione per tutte le chiamate ricorsive, utilizzando l'assegnazione per modificare i valori dei parametri della funzione su ogni chiamata.

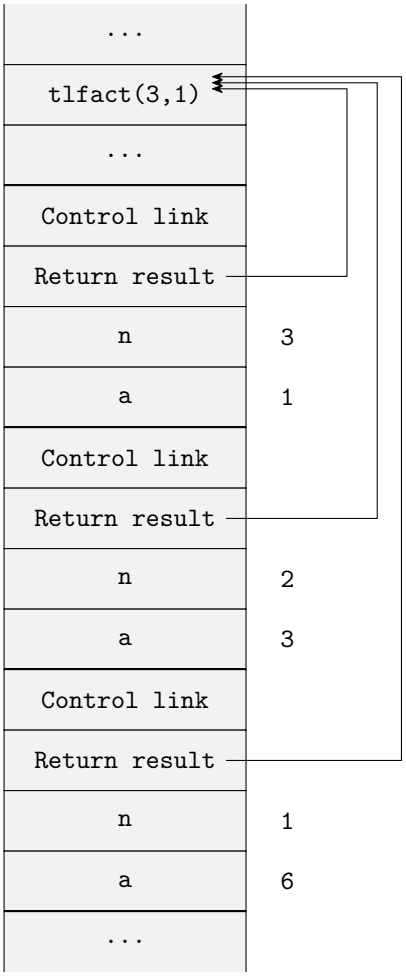


Figura 1.1: Senza tail recursion

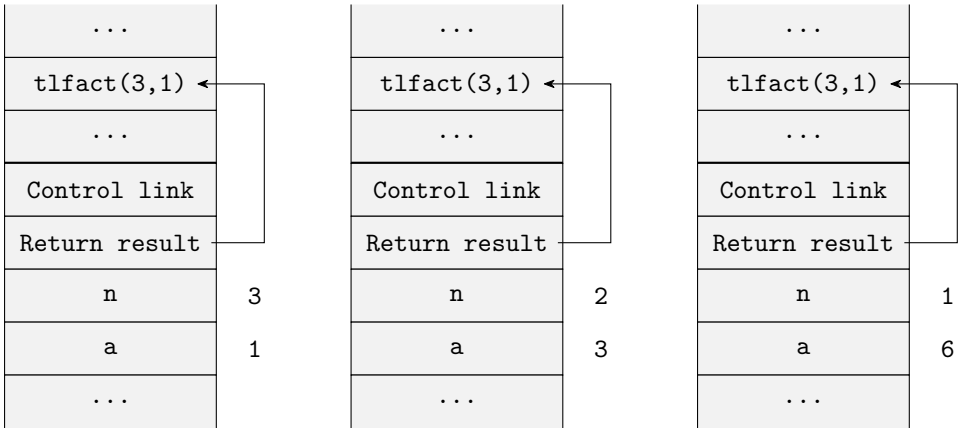


Figura 1.2: Ottimizzazione grazie alla tail recursion

Ricorsione della coda come iterazione

Quando viene compilata una funzione ricorsiva di coda, è possibile generare codice che calcola il valore della funzione mediante un ciclo iterativo. Quando viene utilizzata questa ottimizzazione, le funzioni ricorsive della coda possono essere valutate da un solo record di attivazione per una prima chiamata e tutto il risultato chiamate ricorsive, non un record di attivazione aggiuntivo per chiamata ricorsiva. Ad esempio, la precedente funzione fattoriale ricorsiva della coda può essere compilata nello stesso codice di quella della seguente funzione iterativa:

```
fun itfact (n, a) =
        while n> 0 do
                a: = n * a;
                n: = n-1;
        endwhile
        a;
```

Un record di attivazione per `itfact` sembra proprio un record di attivazione per `tlfact`. Se guardiamo al valori di n e a su ogni iterazione del ciclo, troviamo che cambiano esattamente nello stesso il modo è per le chiamate ricorsive di coda a tlfact. Le due funzioni calcolano esattamente lo stesso risultato la stessa sequenza di istruzioni. Detto in altro modo, l'eliminazione della ricorsione della coda compila la coda funzioni ricorsive in loop iterativi.

2 Type safety

La sicurezza dei *tipi* in un programma è molto importante. Se l'esecutore non riesce a distinguere i tipi di un certo programma può facilmente causare errori. Molti attacchi sfruttano proprio debolezze nel controllo dei tipi di linguaggi diffusi come il C.

> Well-typed programs never go wrong.
> — *Robert Milner*

2.1 Tipo

Un tipo è un insieme di valori omogenei più le operazioni che si possono fare.

Esempi di tipi:

tipi semplici `Boolean`, `Integer`, `String`, ...

tipi strutturati Classi, ...

Anche le funzioni definiscono un tipo, per esempio una funzione che riceve un int e ritorna un bool.

Esempi di non tipi:

- numeri dispari;
- array contenenti `String` e `Integer`.

Può però dipendere dal linguaggio di programmazione.

2.1.1 Utilizzo

I tipi servono per:

- organizzare e dare un nome ai concetti (documentazione)
  - indicare l'uso che si vorrà fare di certi identificatori (così il compilatore può controllare)
- assicurarsi che sequenze di bit in memoria siano interpretate correttamente
  - per evitare errori come assegnare ad una variabile `3 + true + "Angelo"`
- istruire il compilatore come rappresentare i dati
  - per memorizzare il tipo `short` servono meno bit rispetto al tipo `int`

2.2 Errori

2.2.1 Possibili errori

Quali errori si possono commettere?

- Errori di tipi a livello hardware
  - Confondere dati con programmi, caricando quindi nei registri della CPU codici non corretti
    - Esempio: eseguo `x()` ma in realtà `x` non è una procedura ma un intero.
  - Confondere tipi di dati semplici
    - Esempio: eseguo `float_add(3, 4.5)` con il primo parametro intero. Se la CPU prende il numero 3 come sequenza di bit `float`, potrebbe generare un errore hardware
- Errori semantici (il programma fa qualcosa che non dovrebbe)
  - Esempio (tipi primitivi)

    `int_add(3, 4.5)` in questo caso la sequenza di bit che rappresenta il numero `4.5` può essere interpretato come intero modificandone il valore (arrotondandolo)
  - Esempio (oggetti)

```
class Quadrato extends Figura {}

// Corretto
Figura a = new Quadrato();

// Sbagliato: Quadrato potrebbe avere dei metodi in
    piu'
// rispetto a Figura che potrei invocare ma non
    trovare
Quadrato b = new Figura();
```

2.3 Sicurezza dei tipi

Ora che sappiamo cosa è un *tipo*, quali *errori* si possono commettere e quindi abbiamo compreso l'importanza della *sicurezza dei tipi* definiamo L un linguaggio di programmazione **type safe** se non esiste programma scritto in L che possa violare la distinzione di tipi in L.

2.4 Type checking

I linguaggi possono essere classificati anche in base al momento in cui verificano la correttezza del codice.

Tabella 2.1: Type-safety di alcuni tra i linguaggi più diffusi.

Type-safety	Linguaggi	Motivo
Assente	C, C++	Unchecked type cast, aritmetica dei puntatori
Parziale	Pascal	Deallocazione esplicita e dangling pointers
Completa	Java, Lisp, Python	Controllo completo dei tipi

2.4.1 Run-time type checking

Il controllo sui tipi avviene durante l'esecuzione del programma

Esempio: Lisp quando esegue l'istruzione `(car x)` (applica `car` alla lista `x`, restituendone il primo elemento) controlla prima che x sia una lista.

2.4.2 Compile-time type checking

Il controllo avviene durante la compilazione

Esempio: ML se compila `f(x)` controlla che se `f` sia del tipo `A -> B` e che `x` sia del tipo `A`.

2.4.3 Pro e contro

- Entrambi gli approcci prevengono gli errori di tipo
- Run-time checking rallenta l'esecuzione (controlla le conversioni di tipo ogni volta)
- Compile-time checking limita la flessibilità dei programmi (tutte le istruzioni, anche quelle che non verranno mai eseguite, devono essere corrette, perché il controllo dev'essere *conservativo)*

Alcuni programmi che non sono corretti compile-time sono invece corretti run-time:

```
int x = 0, y;
if (x == 0)  // Sempre vera, eseguo solo il ramo if
  y = 5;
else
  y = true;  // Assegnazione scorretta (int <- bool)
             // ma mai eseguita
```

2.5 Type Checking in Python

In python ogni variabile ha un tipo e viene controllata (type safe) ma: Non è necessario dichiarare il tipo di una variabile

```
x = 5
print(type(x))
```

Posso comunque specificare il tipo (con ancge conversioni esplicite)

```
x = float(20)
x = bool(5)
z = float("3")
```

Controlla comunque che un'operazione sia type safe:

```
y = x / "pippo" # errore idi tipo perché non esiste l'
    opereazione / tra int e string
```

I tipi si possono ridefinire

```
x = int(5)
print(type(x))
x = "pippo"
print(type(x))
```

Problemi Se il mio codice contiene un errore di tipo e non lo eseguo non me ne accorgo

```
x = int(5)
if x < 5:
y = x / "pippo"
print(x)
```

Attenzione con la definizione di funzioni:

```
# define a sum function (intended to be used for integers)
def sum(x,y):
        return x+y

print( sum(8,'hello')) # dará errore solo quando chiamata
```

Puoi usare mypy:

```
def sum(x:int,y:int) -> int:
        return x+y
print(sum(8,4))
```

2.6 Problemi dei vari linguaggi

2.6.1 C/C++

Questi linguaggi hanno un sistema di tipi non sicuro, posso violare facilmente la distinzione di tipi. Gli errori più tipici:

- unchecked type cast;
- dereferenziazione del `null`;
- pointer arithmetic;
- accesso a memoria non valida (out of bound e dangling pointer).

2.6.2 Java

Java usa compile-time checking, ma dove il compilatore non è sicuro della sicurezza dei tipi, introduce un controllo run-time, per esempio:

```
Quadrato a = (Quadrato) b;
```

- `b` è dichiarato precedentemente di tipo `Figura` (estesa da `Quadrato`);
- la conversione a `Quadrato` è corretta solo se `b` è effettivamente un'istanza di `Quadrato` (o di una sua sottoclasse);
- quest'ultimo controllo può essere fatto solo in tempo di esecuzione.

2.7 Problemi del C

Type cast non safe

Il C permette la conversione *non controllata* da un tipo ad un altro:

- da un tipo ad un altro con possibile perdita di informazioni:

  ```
  double d = 10.3;
  int i;
  i = d;
  // Programma corretto, ma i conterrá il valore intero 10,
  // quindi ho una perdita di informazioni
  ```

- da un intero ad una funzione, ovvero cerco di eseguire una certa locazione di memoria che potrebbe non essere un'istruzione corretta e fare quindi qualcosa di non voluto.

Deferenziazione di null

La deferenziazione di un puntatore in C non viene controllata. Se accedo ad una cella puntata da un puntatore nullo ho l'errore *segmentation fault*, ovvero un errore di accesso alla memoria gestita dal sistema operativo.

```
int main() {
  int *ptr;
  ptr = NULL;
  *ptr = 2;     // Errore
}
```

Pointer arithmetic

Mediante l'aritmetica dei puntatori possiamo puntare a zone di memoria con tipi diversi, per esempio:

1. dichiaro un puntatore `p` di tipo `A*` e una variabile `x` sempre di tipo `A`;
2. l'espressione `*(p+i)` ha tipo `A`;

3. ma in memoria il valore memorizzato in posizione `p+i` potrebbe avere qualsiasi tipo;

4. quindi `x = *(p+i)` permette di memorizzare un valore di qualsiasi tipo in `x`.

Memory safe

Il C non è memory safe. Posso, mediante i puntatori, accedere alla memoria in modo scorretto. Per esempio:

```
void foo (int *p, int i, int v) {
  p[i] = v;
}
```

Con questa funzione accedo all'indirizzo `(p+i)` che potrebbe contenere dati importanti (o altro codice); per esempio potrebbe contenere il *return address* di una chiamata di una procedura o potrei modificare dei diritti e/o leggere informazioni riservate.

Un esempio tipico dell'errore appena descritto è il cosiddetto **buffer overflow** (o buffer overrun).

Questo errore succede quando un buffer dichiarato sullo stack viene sovrascritto copiando in esso un dato più lungo. Ciò comporta la sovrascrittura di celle di memoria consecutive al buffer che potrebbero contenere dati importanti come il return address.

```
void foo (char *bar) {
  char c[10];
  strcpy(c,bar);
  // La lunghezza di bar non é nota e potrebbe superare
  // quella di c
}

int main (int argc, char *argv[]) {
  foo(argv[1]);
}
```

Type cast e violazione memoria

I puntatori in C sono assimilati a interi, tramite cast di dati interi a puntatori posso accedere ad una zona di memoria a piacere:

```
int main() {
  char *ptr;
  ptr = 1000;  // Conversione da int a char*
  *ptr = 'a';
}
```

Violazione di memoria con stringhe

Perchè il C ha NUL terminated strings?

The Most Expensive One-byte Mistake
Did Ken, Dennis, and Brian choose wrong with NUL-terminated text strings?
by Poul-Henning Kamp | July 25, 2011,ACM
http://queue.acm.org/detail.cfm?id=2010365
...one of the main causes of the fall of the Roman Empire was that, lacking zero, they had no way to indicate successful termination of their C programs.
Robert Firth

Deallocazione esplicita e Dangling pointers

In pascal, C, etc., una locazione puntata da un puntatore `p` può essere deallocata: a questo punto `p` è un *dangling pointer*. Ad esempio in C, posso fare il `free` di un puntatore e poi continuare ad usarlo.

Un puntatore è *dangling* se punta ad una zona di memoria che è stata liberata per essere riutilizzata.

Il sistema operativo potrebbe allocare la stessa memoria nuovamente per memorizzare un altro tipo di valore. Posso continuare ad usare `p` per accedere a questa memoria e rompere la type safety.

Esempio:

```
char *itoa(int i) {
  char buf[20];
  sprintf(buf, "%d", i);

  // buf viene restituito ma punta ad un array locale
  // nello stack, che viene deallocato al termine della
     funzione
  return buf;
}
```

Dangling pointers sullo stack

Esempio in C++:

```
struct Point { int x, y; };

Point *makePoint(int x, int y) {
  Point result = {x, y};
  return &result;              // Puntatore ad un oggetto
     locale!
}

void bar() {
  Point *p = makePoint(1,2);  // Dangling pointer
  p.y = 1234;
}
```

Soluzione al dangling pointer

Come evitarlo?

1. Evitare di puntare zone di memoria sullo stack ed usare la malloc:

   ```
   Point *result = (Point*) malloc(sizeof(Point));
   ```

   La malloc crea puntatori a zone sicure, però la sua gestione non è automatica come le variabili sullo stack.

2. Uso del garbage collector (gc) invece che della deallocazione esplicita:
   - Il *gc* marca le zone da liberare e che si possono riutilizzare.
   - Non usando free, il *gc* recupera la memoria.

2.7.1 Soluzioni agli errori

Se vogliamo scrivere codice safe cosa possiamo fare?

Scrivere attentamente, anticipare al codice una approfondita progettazione e documentazione, ecc...

Se vogliamo essere sicuri che il nostro codice è safe cosa possiamo fare?

- usare linguaggi type safe (Java, lisp, ...) e linguaggi più astratti;
- usare linguaggi come C e dei tools che ci aiutino a rendere i programmi safe.

Svantaggi ad usare linguaggi astratti

C'è un prezzo da pagare se si vogliono usare linguaggi safe come Java e Lisp:

- prestazioni inferiori, per il controllo dei limiti nell'accesso agli array e per la garbage collection che evita dangling pointers;
- impiego di maggiore memoria, per tenere informazione sui tipi, sulla dimensione degli array, ...;
- annotazione dei tipi, maggiore verbosità nelle dichiarazioni;
- porting di codice già esistente in C, per quanto abbiano sintassi simile.

Vantaggi del C

Il C è tutt'oggi usato per molte applicazioni come sistemi operativi, device drivers, ecc. poiché:

- ha prestazioni elevate;
- permette la gestione esplicita della memoria;
- permette il controllo della rappresentazione dei dati a basso livello;
- esiste molto codice scritto in C, che può essere riutilizzato.

Come rendere il C safe

1. Usando dei tools per l'analisi statica e dinamica per trovare le violazioni:
   a) Purify della Rational/IBM è un tool per l'analisi *dinamica* per scoprire errori di accesso alla memoria; esiste anche Valgrind.
   b) Cppcheck, Lint, ... per l'analisi statica;
2. uso di un sottoinsieme del C;
3. usando librerie per rendere i programmi C safe;
4. tools e (sotto)linguaggi per prevenire safety violation con due approcci distinti:
   a) rendere sicuri i programmi scritti in C: SafeC, CCured, ...;
   b) usare varianti safe del C: **Cyclone**, Vault.

1. Uso di tools per trovare violazioni

Splint Secure Programming Lint, è un tool per il controllo statico dei programmi C per la vulnerabilità della sicurezza e gli errori nel codice. Spesso è chiamato LCLint ed è una versione moderna del tool Lint di Unix.

Splint ha la capacità di interpretare delle annotazioni speciali del codice sorgente, questo dà un controllo più forte di quanto sia possibile farlo controllando solamente il codice.

L'ultima versione risale al 2007.

Purify Si approccia con una analisi dinamica, riceve in input un qualsiasi programma in C/C++ e manda in output programmi eseguibili linkati con Purify.

Come funziona? Inserisce dei controlli per trovare durante l'esecuzione errori di accesso alla memoria o di memoria non rilasciata.

Aspetti positivi: si applica a codice già esistente.

Aspetti negativi: Rallenta l'esecuzione e non garantisce la scoperta di ogni errore.

Valgrind Effettua un'analisi dinamica (come Purify).

Valgrind attualmente comprende sei production-quality tools:

- Un rilevatore di errori di memoria.
- Due rilevatori di errori thread.
- Una cache e un branch-prediction profiler, una call-graph che genera i primi due componenti e un heap profiler. E sono anche inclusi tre tools sperimentali: un heap/stack/global array per la rilevazione degli overrun, un secondo heap profile che esamina come i blocchi dello heap siano usati and a SimPoint basic block vector generator.
- È eseguibile su X86/Linux, AMD64/Linux, ARM/Linux, PPC32/Linux, PPC64/Linux, X86/Darwin and AMD64/Darwin (Mac OS X 10.5 and 10.6).
- È open source.

2. Uso di un sottoinsieme del C

Ad esempio si usa MISRA C.

3. Librerie safe per le stringhe

Lo scopo dell'utilizzo di queste librerie è quello di evitare i buffer overflow e altri problemi tipici delle stringhe e dei buffer di `char`.

Safe C String Library

ISO/IEC TR 24731

SafeC Prende in input programmi C qualsiasi e restituisce in output programmi C sicuri.

Come fa? Garantisce la cattura delle violazioni di memoria e vari errori run-time inserendo dei controlli e aggiungendo informazioni (ad esempio ai puntatori).

Aspetti positivi: si applica a codice già esistente scritto in C.

Aspetti negativi: rallenta l'esecuzione e aumenta la memoria.

CCured Prende in input programmi C con annotazioni particolari (ma anche senza) e restituisce in output programmi C sicuri.

Come fa? Abbina l'analisi statica a controlli dinamici e garbage collector.

Aspetti positivi: Si applica a codice C già esistente con minime modifiche.

Aspetti negativi: Rallenta l'esecuzione.

Cyclone Prende in input programmi in C modificati e restituisce in output programmi in C sicuri.

Come fa? Applica controlli dinamici solo dove necessario e garbage collector.

Aspetti positivi: Minimo overhead di tempo e di memoria.

Aspetti negativi: Richiede di modificare i programmi originali.

Verrà ampiamente approfondito nel capitolo successivo.

Vault Prende in input programmi in C modificati e restituisce in output programmi COM object.

Come fa? Utilizza un linguaggio astratto simile a Java/C#.

Aspetti positivi: minimo overhead di tempo e di memoria.

Aspetti negativi: Richiede di riscrivere i programmi originali.

4. Uso di altri linguaggi C-like

Controllo sui dettagli a basso livello:

- SafeC e CCured: pieno utilizzo del C, operazioni limitate sui puntatori.
- Cyclone: più restrittivo del C.

- Vault: meno efficiente, più astratto.

Opzioni sulla gestione della memoria:

- Cyclone: diverse opzioni.
- Vault: oggetti "lineari" e regioni.
- SafeC: `malloc()` e `free()` esplicite.
- CCured: garbage collection.

Prestazioni e aumento della memoria:

- Cyclone *approx* Vault $<$ CCured $\ll$ SafeC.

Sforzo per annotare i tipi:

- SafeC $<$ CCured $<$ Cyclone $\approx$ Vault.

Sforzo per portare codice esistente:

- SafeC $<$ CCured $<$ Cyclone $\ll$ Vault.

2.8 Cyclone

2.8.1 Materiale, installazione ed esecuzione

Per programmare in Cyclone ci si può affidare a:

- questi appunti;
- Wikipedia;
- sito di Cyclone.

Installazione per Linux dai file sorgenti

1. Scaricare i sorgenti dal sito di Cyclone:
   - Se si usa `gcc` versione 3 si può scaricare il `tar` da `http://cyclone.thelanguage.org/wiki/Download`.
   - Se avete `gcc` versione 4: è meglio prendere l'ultima versione dal repository `svn`:

     ```
     svn --username anonymous co \
       https://source.seas.harvard.edu/svn/cyclone
     ```

     Cercare poi sotto `trunk`.
2. Se si è scaricato il `tar`, scompattare con `tar -zxf`.
3. Aprire una shell e fare `cd` nelle directory dove c'è Cyclone.

4. Da shell: `./configure`.

   Se vuoi metterlo in una directory particolare (esistente), usa

   ```
   ./configure --prefix=/percorso/alla/cartella
   ```

5. `make`.

   (Potrebbe essere necessario installare un vecchio compilatore e fare: `make CC=gcc34`.)

6. `make install`.

Installazione per Linux dai file compilati

Sul sito del corso trovate il pacchetto già pronto per Ubuntu.

Installazione per Windows

Come per Linux dai sorgenti (con `gcc` 3), però:

- installare Cygwin;
- salvarli in una directory senza spazi;
- apri un terminale cygwin e dai i comandi elencati precedentemente per Linux.

Se avete problemi scaricate lo `zip` che contiene tutto.

Installazione per windows dai compilati (`zip`)

- Installa Cygwin (32 bit) al solito;
- copia lo `zip` dal sito `cs` nella directory `C:/Users/c:/cygwin`;
- scompatta lo `zip`.

Eseguire Cyclone

Aggiungi la directory `cyclone/bin` nel tuo `PATH` o chiama `cyclone` con tutto il percorso.

Compiliamo un esempio `hello.cyc` semplicemente eseguendo

```
cyclone -o hello hello.cyc
```

Se l'hai scompattata dallo `zip` dovrebbe essere già nel `PATH`.

2.8.2 Situazione unsafe coi puntatori

Puntatori nulli

```
int main() {
  int *ptr = 0;   // Assegno ad un puntatore NULL un valore
  *ptr = 2;       // Cyclone previene errori inserendo
                  // un controllo ad ogni deferenziazione
}
```

Cast da `int` a puntatore

```
int main() {
  char *ptr;

  // Cyclone vieta questo cast perché permette di
     sovrascrivere
  // la memoria in modo arbitrario
  ptr = 1000;

  *ptr = 'a';
}
```

Salvando questo codice nel file `error1.c`:

```
$ gcc -o error1 error1.c
error_1.c: In function 'main': error_1.c:4:
warning: assignment makes pointer from integer without a cast
$ ./error1
Segmentation fault
```

Aritmetica dei puntatori

```
int main() {
  int x[100];

  // In Cyclone questo non é permesso perché
  // permette di puntare oltre i limiti dell'array
  int *ptr = x;
  ptr += 2000;
  *ptr = 2;
}
```

```
$ gcc -o error2 error2.c
$ ./error2
Segmentation fault
```

2.8.3 La sintassi di Cyclone

Cyclone non permette la compilazione di programmi unsafe!

`NULL` pointer

```
int *ptr;
...
ptr = NULL;
...
*ptr = 2;
```

Abbiamo detto che Cyclone quando ha una deferenziazione inserisce un controllo di non nullità prima dell'accesso al valore puntato.

Ciò è dispendioso. Come posso evitare di inserire sempre un controllo di non nullità? Ad esempio se sono sicuro che il puntatore non sarà mai `NULL`?

Cyclone introduce un nuovo tipo di puntatore, un puntatore non-`NULL`, indicato con `@notnull` oppure brevemente mettendo `@` al posto di `*`:

```
int * @notnull ptrA;
int @ ptrB;
```

Un puntatore `@notnull` non è mai `NULL`, Cyclone controlla quando lo usi (non puoi assegnargli `NULL`) quindi quando accedi al suo contenuto Cyclone può evitare di controllare che sia non nullo.

```
...
FILE *f = fopen("/pwd","r");
int c = getc((FILE * @notnull)
    f);
...
```

```
...
FILE @f =  (FILE@) fopen("/pwd
    ","r");
int c = getc(f);
...
```

Fat pointer

Per prevenire il problema del buffer overflow Cyclone limita l'aritmetica dei puntatori.

Per fare ciò introduce un puntatore *fat*, che mantiene un'informazione aggiuntiva sulla dimensione dell'array.

I puntatori fat sono denotati come `@fat` o brevemente con `?`.

Esempio in C:

```
int strlen(char *p) {
        // Assumo che il buffer termini con '\0',
        // Pero' se non c'é 0 va avanti oltre la fine del
            buffer
        int i = 0;
        for (; *p != '\0'; i++, p++);
        return i;
}
int main(void) {
        char a[] = "angelo";
        printf("%d\n", strlen(a));  // -> 6 OK
        // Se bin metto terminatore ottego un risultato
            sbagliato.
        char b[] = { 'h', 'e', 'l', 'l', 'o', '!' };
        printf("%d\n", strlen(b)); // -> 12
        // Soluzione?
        // s dovrebbe portarsi dietro la sua dimensione!
        return EXIT_SUCCESS;
}
```

Versione in cyclone:

```
int strlen (const char ?s) {
  int i, n;
  if (!s) return 0;
  // 'numelts restituisce la dimensione dell'array
  n = numelts(s);
  // Quando faccio '*(s+i)' Cyclone inserisce un bound check
  for (i = 0; i < n; i++, s++) {
```

```
    if (*(s+i) == '\0')
      return i;
  }
  return n;
}
```

Zero-termination

Per rappresentare le stringhe si può usare questo tipo di puntatori: `char * @zeroterm`.

Questo particolare puntatore può essere combinato con i puntatori *notnull* e *fat*.

Di default un puntatore è `@nozeroterm` (ad esclusione delle stringhe). Quando si esegue `(x+i)` Cyclone controlla che tra `x` e `x+i-1` non vi siano `NULL`.

L'utilizzo può diventare pesante computazionalmente.

Creazione Array

```
// La vita di questi array coincide con quella del
// record di attivazione sullo stack
int foo[8] = {1, 2, 3, 4, 5, 6, 7, 8};
char s[4] = "bar";
```

Per creare un array allocato sullo heap:

```
int * @numelts(8) foo = new {1, 2, 3, 4, 5, 6, 7, 8};
char * @fat s = new {'b', 'a', 'r', '\0'};
```

Conversioni

- Gli array possono essere convertiti in puntatori di tipo `?`:

  ```
  char a[7] = "Angelo";
  char * @fat afat = a;
  strlen(afat);
  ```

- Si possono convertire puntatori `*` a `?` con size `1`.

  Che vuol dire? Che il compilatore accetta l'aritmetica sui `*`, convertendoli in `?` con size `1` e poi a run-time controlla se si accede "out-of-bound".

- Si possono convertire puntatori `?` a `*`.

  Il casting da `?` a `*` fa invocare il controllo dei limiti, mentre il casting da `?` a `@` fa invocare sia il controllo sul puntatore `NULL` che il controllo sui limiti.

Esempio con i vettori:

```
void foo(int *@ numelts(4) arr);

int y[8] = {1, 2, 3, 4, 5, 6, 7, 8};

// Il tipo di y viene automaticamente convertito nel tipo
// 'int *@ numelts(8)', e siccome 8 >= 4
// Cyclone accetta (con un warning)
```

```
foo(y);

// Questa seconda parte del codice viene invece rifiutata
// in quanto 'bad' é troppo corto
int bad[2] = {1, 2};
foo(bad);
```

3 Programmazione orientata agli oggetti

3.1 Modularità

Uno dei concetti base per l'ottima progettazione di un sistema software è la *modularità*; infatti, un codice può essere diviso in più moduli totalmente indipendenti sia nella fase di sviluppo che di test. Secondo questo principio ogni modulo può essere sostituito senza intaccare in alcun modo le altre parti del sistema, mantenendo semplicemente la medesima interfaccia del precedente.

In un articolo molto influente del 1969 intitolato "Structured Programming", E.W. Dijkstra argomentò come un programma debba essere sviluppato inizialmente delineando le funzioni principali che deve implementare e rifinendo poi queste funzioni come aggregati di sottofunzioni minori, procedendo in questa maniera iterativamente fino a raggiungere un livello tale che le funzioni delineate possono essere espresse come semplici operazioni base. Questo procedimento produce sottoproblemi piccoli abbastanza per essere compresi e separati abbastanza per essere risolti indipendentemente.

D'altronde le parole di Dijkstra sono solamente l'applicazione diretta del principio *Divide et Impera*. Questo concetto in informatica prende il nome di "progettazione top-down" e si applica perfettamente all'idea di modularità che stiamo descrivendo.

Due concetti fondamentali per la programmazione modulare, quindi, sono le interfacce e le specifiche:

Interfaccia Descrizione delle parti di un componente che sono visibili agli altri componenti del programma.

Specifica Descrizione del comportamento che un componente dovrebbe assumere, osservato attraverso la sua interfaccia.

Ad esempio, nel caso di una funzione la sua interfaccia è composta dal nome, dal numero e tipo di parametri che riceve in input e dal parametro di ritorno; la sua specifica, invece, descrive la relazione tra i parametri in ingresso e quello di ritorno.

3.2 Astrazione dei dati

Un'altra proprietà molto importante per i linguaggi di programmazione è l'*astrazione*, un meccanismo in grado di enfatizzare le proprietà generali di un segmento di codice nascondendone i dettagli all'utente finale. A questo concetto è spesso associato quello di *information hiding* (incapsulamento), il quale consiste nel nascondere a chi utilizza determinati dati come questi vengono elaborati e ottenuti.

Una approccio di sviluppo modulare facilita l'astrazione dei dati e riassume il loro funzionamento alle semplici interazioni specificate dalle interfacce. Quando un programma è scritto con un approccio di questo tipo, quindi, è possibile cambiare la

struttura interna di ogni componente in maniera trasparente al resto del programma, a patto che il comportamento visto attraverso la sua interfaccia non cambi.

In linea teorica, una funzione che non utilizza variabili globali permette modularità, astrazione, information hiding e riutilizzo del codice.

L'astrazione dei dati si pone tre obiettivi principali:

1. identificare l'interfaccia di una struttura dati;
2. provvedere all'information hiding separando le decisioni di implementazione dalle parti di programma che utilizzano la struttura dati;
3. permettere alla struttura dati di essere utilizzata in diversi modi e da diversi programmi.

Più in generale, il meccanismo di un tipo di dato astratto (*Abstract Data-Type*) associa il tipo di un dato con una struttura dati avente un set di funzioni (interfaccia) che ha accesso diretto ai dati, ma che non permette alle altre parti del programma di accedervi se non attraverso queste funzioni. Dunque, gli abstract-data-type non permettono né riuso di codice né a componenti aventi la stessa interfaccia di essere sostituite tra loro.

3.3 Concetti base della programmazione orientata agli oggetti

Un oggetto consiste in una serie di operazioni che possono essere eseguite su un set di dati nascosto. I principi di modularità ed astrazione dei dati appena descritti, quindi, sono basilari nella programmazione a oggetti (*object-oriented*) e sono tra i motivi principali del suo sviluppo.

La progettazione a oggetti si evolve identificando i concetti principali di un problema e utilizza gli oggetti per rappresentarli all'interno del sistema software. Dunque, i passi principali da seguire per la realizzazione di un sistema object-oriented sono:

- identificare gli oggetti ad uno specifico livello di astrazione;
- identificare il comportamento di questi oggetti;
- identificare le relazioni tra gli oggetti;
- implementare gli oggetti.

In un oggetto sono definiti *metodi* le funzioni che lo compongono, mentre i dati che ne fanno parte sono solitamente chiamati *campi*.

I quattro principi base su cui sono progettati i linguaggi orientati agli oggetti sono i seguenti:

Abstraction (Astrazione) I dettagli dell'implementazione sono nascosti nel codice da un'interfaccia, la quale per gli oggetti consiste solitamente in un set di funzioni predisposte per limitare l'accesso ai dati.

Subtyping (Sottotipazione) Se un oggetto `a` implementa tutte le funzionalità di un oggetto `b` allora possiamo utilizzare `a` in ogni contesto in cui ci aspettiamo di usare `b`.

Inheritance (Ereditarietà) Abilità di riutilizzare la definizione di un oggetto per definire un altro oggetto.

Dynamic lookup (Risoluzione dinamica o binding dinamico) La risposta ad un messaggio (o chiamata di funzione) ricevuto dell'oggetto dipende dal modo in cui quest'ultimo è implementato, e quindi oggetti diversi possono rispondere allo stesso messaggio in maniera diversa.

3.3.1 Dynamic lookup

Come già descritto questa caratteristica dei linguaggi di programmazione orientati ad oggetti prevede che la reazione ad una chiamata di funzione effettuata su di un particolare oggetto dipende proprio dalla tipologia di oggetto di cui si tratta.

Supponiamo infatti di effettuare una somma su di un oggetto x, se l'istanza di questo oggetto al momento della chiamata della funzione è di tipo intero la chiamata effettuerà una somma algebrica, se invece in un secondo momento a x viene assegnato un oggetto di tipo stringa, la stessa riga di codice potrebbe causare una concatenazione.

Questo principio non va confuso con l'overloading, il quale prevede di definire con lo stesso nome diverse funzioni all'interno dello stesso oggetto e viene risolto in fase di compilazione; mentre il dynamic lookup, invece, prevede l'utilizzo di oggetti differenti a chiamate differenti in base al contesto dinamico su cui si è sviluppato il programma e dunque viene definito a runtime.

Questa caratteristica permette di generalizzare particolari funzioni rendendole indipendenti dagli oggetti che utilizzano. Basti pensare ad un programma di grafica che utilizza funzioni come "disegna" o "cancella" con riferimento a diverse forme, assegnando semplicemente in maniera dinamica le risposte del programma all'oggetto utilizzato.

3.3.2 Abstraction

Applica tutti i principi di astrazione dei dati studiati nei capitoli precedenti; nasconde i dati dell'oggetto e la sua implementazione, riducendone gli accessi a quanto specificato dalla sua interfaccia, e cioè dal set di funzioni predisposte per accedere ai dati.

3.3.3 Subtyping

Consiste nella relazione tra le interfacce di più oggetti che permette ad una particolare tipologia di oggetto di essere utilizzata al posto di un'altra in particolari condizioni. Infatti, *se l'insieme di funzioni messe a disposizione da un oggetto `B` (o interfaccia di `B`) contiene l'interfaccia di `A` (quindi l'interfaccia di `A` è sottoinsieme dell'interfaccia di `A`), allora `B` è sottotipo di `A`.*

Il principio di *polimorfismo*, inoltre, stabilisce che se `B` è sottotipo di `A` allora posso utilizzare ogni volta un oggetto di tipo `B` in sostituzione ad uno di tipo `A`.

Un primo vantaggio della sottotipazione è che permette di eseguire le stesse operazioni su diverse tipologie di oggetti, sottotipi però di uno stesso oggetto comune che implementi quelle operazioni.

3.3.4 Inheritance

Consiste nella relazione tra le implementazioni di un oggetto che permette di definire una nuova istanza riutilizzando codice già scritto in precedenza per oggetti già esistenti.

In particolare permette ad un oggetto `B` "figlio" di un oggetto `A` "padre" di ereditare il codice di determinate funzioni (ridefinendone magari altre) senza bisogno di effettuare duplicazione di codice, ottimizzando la gestione di memoria e facilitando i meccanismi di manutenzione dei programmi.

Si vengono quindi a creare due interfacce all'interno di un oggetto: una pubblica, a disposizione dell'utente finale, ed una protetta, che mette a disposizione parti di codice che possono essere ereditate delle classi figlie.

L'ereditarietà tra un oggetto `B` figlio di un oggetto `A` ammette che:

- `A` possa nascondere del codice a `B` (dati o funzioni)
- `B` possa introdurre nuovi metodi
- `B` possa ridefinire alcuni metodi di `A`
- `B` possa nascondere alcuni metodi ad `A`

La differenza tra ereditarietà e sottotipazione sta proprio nella loro definizione. Infatti, se un oggetto `B` è sottotipo di un altro oggetto `A` significa che `B` può essere utilizzato al posto di `A`, mentre se un oggetto `B` è figlio di un altro oggetto `A` significa che `B` potrebbe utilizzare codice definito nell'implementazione di `A`.

Per comprendere meglio questa distinzione (spesso mescolata nell'implementazione di alcuni linguaggi) pensiamo all'esempio di una possibile rappresentazione delle strutture dati di coda, pila e coda bilaterale (da cui posso inserire sia in "in testa" che "in coda" e posso estrarre "in coda"). Una coda può essere implementata a partire dal codice di una coda bilaterale semplicemente limitandone l'interfaccia all'inserimento "in testa" e all'estrazione "in coda". Similmente anche per la pila se consideriamo solo l'inserimento "in coda" e l'estrazione "in coda". Nonostante ciò non possiamo definire né la coda né la pila sottotipo di una coda bilaterale siccome non possono essere utilizzate in sua vece. Infatti, se una funzione f richiede in ingresso una coda bilaterale non potremo al suo posto passare una pila o una coda. Questo esempio dimostra chiaramente che nonostante sia pila che coda possono essere figlie della coda bilaterale, quest'ultima è sottotipo delle prime due, il che distingue nettamente i concetti di ereditarietà e sottotipazione.

3.4 Information hiding in C

In questo paragrafo vengono illustrati diversi metodi per applicare il principio di incapsulamento nel linguaggio C. Ovviamente questo linguaggio non è object-oriented e quindi l'utilizzo di questi particolari costrutti non è agevolato da particolari strutture, ma è tuttavia possibile sfruttarlo in parte attraverso alcuni artifici.

In tutti i metodi illustrati è necessaria la divisione tra file header (`*.h`) e file principale (`*.c`). Per facilitarne la spiegazione abbiamo preso come esempio un ipotetico oggetto contatore. Del contatore ci interessano tre operazioni:

1. reset: per azzerare un contatore
2. inc: per incrementare il suo valore
3. getValue: per poter recuperare il suo valore

Nello header metteremo le dichiarazione delle funzioni (e se necessario altro), mentre nel file principale metteremo le definizioni. In questo modo, che vuole usare le funzionalità importerà solo il .h e il .c verrà unito solo in fase di linking. Potremo cambiare il .c senza che i clienti del .h se ne accorgano (information hiding).

3.4.1 Primo metodo - uso typedef

Il primo metodo consiste nel ridefinire, attraverso la keyword `typedef`, una particolare tipo di dato dandogli un altro nome a nostra scelta:

```
/* counter.h */
typedef int counter;
void reset(counter*);
void inc(counter*);
int  get_value(counter);

/* counter.c */
#include "counter.h"
void reset(counter *c)    { *c = 0; }
void inc(counter *c)      { (*c)++; }
int  get_value(counter c) { return c; }
```

Per utilizzare questo nuovo dato appena definito in un programma sarà sufficiente includere il file header e chiamare i metodi come illustrato di seguito:

```
/* client.c */
#include "counter.h"

int main() {
  int v1, v2;
  counter c1, c2;
  reset(&c1); reset(&c2);
  inc(&c1); inc(&c1); inc(&c2);
  v1 = get_value(c1);
  v2 = get_value(c2);
  return 0;
}
```

Questo metodo permette di separare l'interfaccia dalla rappresentazione nascondendo parzialmente i dati all'utente finale. Tuttavia, quest'ultimo potrebbe comunque agire sul dato modificandolo senza rispettare i vincoli del contatore:

```
counter c1;
c1 = 1000;
```

3.4.2 Secondo metodo uso di varibili di modulo

Possiamo usare una variabile globale di modulo (definita con `static` keyword) e modificare quella con delle funzioni opportune.

```
/* mcounter.h */
void reset();
void inc();
int  get_value();

/* mcounter.c */
#include "mcounter.h"
static int count;
void reset() { count = 0; }
...

/* mclient.c */
#include "mcounter.h"
int main() {
  int v;
  reset();
  ...
  v = get_value();
  ...
}
```

Questo metodo, a differenza del precedente, rende totalmente inaccessibile il dato all'utente finale, il quale avrà però a disposizione solamente una risorsa.

Nota che se non mettessi `static` la variabile globale di modulo, potrei comunque vederla anche in altri file mediante il costrutto `extern` che permette di riferirsi ad un simbolo definito in altro file senza importarlo. Le variabili statiche, invece, sono visibili esclusivamente nel file dove sono dichiarate.

3.4.3 Abstract data type in C

In questo paragrafo, similmente a quanto fatto nel precedente, studieremo due metodi per utilizzare strutture dati simili agli abstract data type nel linguaggio C.

Useremo un tipo opaco che in C si può fare con i record opachi e i puntatori opachi. Il puntatore opaco è un puntatore che punta ad un record (`struct`) che però non è ancora definito.

Primo metodo - record opaco

Col primo modo introduciamo un tipo puntatore ad una struttura che viene definita solo nel .c.

```
/* counter.h */
typedef struct counter *counter_ref;
counter_ref make_counter();
void        inc_counter(counter_ref);
int         get_value(counter_ref);
void        delete_counter(counter_ref);
```

```
/* counter.c */

#include "counter.h"
#include <stdlib.h>  // malloc, free

struct counter { unsigned long value; }

counter_ref make_counter() {
  counter_ref res = malloc(sizeof(struct counter));
  res->value = 0;
  return res;
}

void inc_counter(counter_ref c)    { (c->value)++; }
int  get_value(counter_ref c)      { return c->value; }
void delete_counter(counter_ref c) { free(counter_ref); }
```

Quando voglio usare la struttura devo crearla, poi posso usarla e alla fine distruggerla.

```
/* usocounter.c */
#include "counter.h"
#include <stdio.h>

int main(){
        counter_ref cr = make_counter();
        inc_counter(cr);
        printf("%d",get_value(cr));
        delete_counter(cr);
}
```

Uso di puntatore generico

Potrei introdurre il puntatore opaco come puntatore generico (`void*`).

```
typedef void *counter_ref;

counter_ref make_counter();
int         get_value(counter_ref);
...

/* adt2.c */

#include "adt2.h"
#include <stdlib.h>  // malloc, free

typedef struct { unsigned long value; } counter_struct;

counter_ref make_counter() {
  counter_struct *ptr = malloc(sizeof(counter_struct));
  ptr-> value = 0;
  return (counter_ref)ptr;
}
```

```
int get_value(counter_ref) {
  return ((counter_struct*)c)->value;
}
...
```

3.5 Design Pattern

Quando si è cominciato a lavorare seriamente con i linguaggi ad oggetti, gli addetti ai lavori si sono resi conto presto che, durante la stesura di un programma, si presentavano spesso dei problemi ricorrenti.

Per questo motivo un gruppo di 4 programmatori, chiamato la banda dei 4 (GoF — Gang of Four), ha cercato di formulare una lista dei 23 problemi più ricorrenti e delle loro soluzioni; fu così che nel 1994 nacquero i *Design Patterns*.

Di seguito studieremo solamente tre delle soluzioni studiate e proposte dalla GoF. Per farlo sfrutteremo esempi scritti in codice Java.

Per comprendere pienamente questo capitolo è caldamente consigliato provare ad utilizzare i pattern qui descritti in programmi creati da voi per esercizio.

3.5.1 Singleton

Il desgin pattern *Singleton* è un pattern creazionale utilizzato in quei casi in cui è necessario esista *solamente un'istanza* di un determinato oggetto, ed è quindi consigliabile che questa restrizione sia implementata rigidamente nella classe di riferimento piuttosto che lasciarla nelle mani del programmatore finale.

Implementare questo design pattern è abbastanza semplice, basta nascondere all'utilizzatore finale il costruttore della classe, rendendolo disponibile solo alle classi figlie attraverso una visibilità `private` (o al più `protected` se si vogliono introdurre sottoclassi, altrimenti non sarà possibile), e l'istanza vera e propria dell'oggetto, rendendola `private`. Dopodiché basta creare un metodo `public` in grado di permettere l'accesso a questa istanza.

Di seguito un codice d'esempio scritto in Java.

```
class Singleton {
  private static Singleton sInstance = null;

  private Singleton() {}

  public static Singleton instance() {
    if (sInstance == null)
      sInstance = new Singleton();

    return sInstance;
  }

  public int doSomething() {
    int res = 0;
    ...
    return res;
  }
```

```
  // Altri metodi
  ...
}
```

3.5.2 Facade

Il design pattern *Facade* è un design pattern strutturale utilizzato per la gestione di oggetti composti da una struttura dati vasta, composta a sua volta da diversi oggetti. Infatti, una classe che implementa questo design pattern si propone di facilitare l'accesso, tramite un singolo oggetto, a un più vasto set di oggetti che lo compongono. Di fatto questo stile architetturale fornisce all'utilizzatore finale un'interfaccia di più alto livello che faciliti l'utilizzo di una serie di componenti accomunate tra loro.

Normalmente un oggetto *Facade* fornisce diversi metodi che ridirigono le richieste dell'utente passandole ai sotto-oggetti adatti.

Di seguito un esempio in linguaggio Java di come potrebbe essere applicato questo design pattern alla gestione di un braccio robotico (l'esempio è puramente didattico).

```
class RobotFacade {
  private Camera c;
  private Braccio b;
  private Pinza p;

  public spostaOggetto(Oggetto o, Posizione pos) {
    Posizione start = c.trovaOggetto(o);
    b.raggiungiPosizione(start);
    p.afferraOggetto();
    b.raggiungiPosizone(pos);
    p.rilasciaOggetto();
  }
}
```

3.5.3 Visitor

Il design pattern *Visitor* è un design pattern ideato per rendere più semplice la manutenzione del codice dedicato alle operazioni da svolgere su determinate classi.

Le classi *Visitor* rappresentano un'operazione da eseguire su un definito set di oggetti e consentono, quindi, di definire nuove operazioni da applicare a questi oggetti senza modificarne le classi. Infatti, capita spesso di dover implementare operazioni distinte e indipendenti su oggetti appartenenti ad una struttura aggregata eterogenea, ma si vuole evitare che le classi siano inquinate con queste operazioni.

Il problema principale di questo approccio deriva dal fatto che alcune operazioni possono non essere applicate a tutti gli oggetti, o avere applicazioni diverse a oggetti diversi. Quindi, si deve scrivere un metodo per ogni classe coinvolta.

Più in generale è necessario utilizzare questo pattern architetturale quando:

- molte operazioni distinte e indipendenti devono essere effettuate sulla medesima struttura, e si vuole evitare di inquinarne la classe con queste operazioni;
- le classi cui è applicato il *Visitor* cambiano raramente, ma spesso si vogliono definire nuove operazioni sulla struttura;

- un oggetto contiene molte classi di oggetti con interfacce differenti.

Di seguito un esempio, scritto in Java di un oggetto *Visitor*, applicato su un oggetto predisposto all'utilizzo dei *Visitor* (*Visitable*).

```
// Interfacce
interface VisitableFigure {
  double accept(FigureVisitor v);
}
interface FigureVisitor {
  double visit(Rectangle r);
  double visit(Circle c);
  double visit(Square s);
}

// Visitables
abstract class Figure implements VisitableFigure {
  public abstract double accept(FigureVisitor v);
}

class Rectangle extends Figure {
  public int base, height;
  ...
  public double accept(FigureVisitor v) {
    return v.visit(this);
  }
}

class Circle extends Figure {
  private int radius;
  ...
  public double accept(FigureVisitor v) {
    return v.visit(this);
  }
}

class Square extends Figure {
  ...
}

// Visitors
class AreaComputer implements FigureVisitor {
  public double visit(Rectangle r) {
    return r.base * r.height;
  }
  public double visit(Circle c) {
    return Math.PI * c.radius * c.radius;
  }
  public double visit(Square s) {
    ...
  }
}

class PerimeterComputer implements FigureVisitor {
```

```
  public double visit(Rectangle r) {
    return (r.base + r.height) * 2;
  }
  public double visit(Circle c) {
    return 2 * Math.PI * c.radius;
  }
  public double visit(Square s) {
    ...
  }
}
```

Questo esempio suppone che il metodo implementato nel *Visitor* restituisca sempre un valore `double`. Tuttavia, è possibile creare anche una versione generica (i generics di Java saranno spiegati nel prossimo capitolo) cui è possibile passare il tipo che deve essere ritornato.

```
// Interfacce
interface VisitableFigure {
  <T> T accept(FigureVisitor<T> v);
}
interface FigureVisitor<T> {
  T visit(Rectangle r);
  T visit(Circle c);
  T visit(Square s);
}

// Visitables
abstract class Figure implements VisitableFigure {
  public abstract <T> T accept(FigureVisitor<T> v);
}

class Rectangle extends Figure {
  public int base, height;
  ...
  public T accept(FigureVisitor<T> v) {
    return v.visit(this);
  }
}

...

// Visitors
class AreaComputer implements FigureVisitor<Double> {
  public Double visit(Rectangle r) {
    return r.base * r.height;
  }
  public Double visit(Circle c) {
    return Math.PI * c.radius * c.radius;
  }
  public Double visit(Square s) {
    ...
  }
}
...
```

4 Java

4.1 Nascita di Java

Il linguaggio di programmazione Java è stato progettato da James Gosling e altri presso Sun Microsystems. Il linguaggio, derivante da un progetto che ha avuto inizio nel 1990, era stato originariamente chiamato Oak ed era stato progettato per essere utilizzato in un dispositivo affettuosamente indicato come un set-top box. Il set-top box era destinato ad essere un piccolo dispositivo di calcolo, collegato a una rete di qualche tipo, e posto sopra ad un televisore. Ci sono varie caratteristiche che un set-top box potrebbe fornire: per esempio, si può immaginare che un browser web venga visualizzato sul televisore e, invece di usare una tastiera, si fa clic sulle icone utilizzando alcuni tasti sul telecomando. Si potrebbe desiderare di selezionare un programma televisivo o un film o scaricare una piccola computer simulation che potrebbe essere eseguita sul dispositivo di calcolo e visualizzata sullo schermo. Una pubblicità televisiva per un'automobile potrebbe consentire di scaricare un tour visivo interattivo dell'automobile, dando ad ogni spettatore una simulazione personalizzata di guidare l'auto sulla strada. Qualunque sia lo scenario considerato, l'ambiente di elaborazione comporterebbe grafica, esecuzione di programmi semplici, e comunicazione tra un sito remoto e un programma eseguito in locale.

Ad un certo punto nello sviluppo di Oak, ingegneri e manager di Sun Microsystems si sono resi conto che c'era un bisogno immediato di un linguaggio di programmazione per browser internet, un linguaggio che potesse essere usato per scrivere applicazioni di piccole dimensioni che possono essere trasmesse attraverso la rete ed eseguite da qualsiasi browser standard su qualsiasi piattaforma standard. La necessità di uno standard è un risultato del desiderio intrinseco di aziende e individui con siti web per poter raggiungere il più ampio pubblico possibile. Oltre alla portabilità, vi è anche un bisogno di sicurezza in modo che qualcuno che scarichi una piccola applicazione, possa eseguirla senza timore di virus informatici o altri pericoli.

Il linguaggio Oak (*Quercia*) è iniziato come una reimplementazione di C++. Anche se il design di un nuovo linguaggio non era un obiettivo finale del progetto, la progettazione del linguaggio è diventata un obiettivo importante del gruppo. Alcune delle ragioni per la progettazione di un nuovo linguaggio sono contenute in questa citazione eccessivamente drammatica, ma comunque informativa, da "The Java Saga" di David Bank in Hot Wired (dicembre 1995):

> Gosling concluse rapidamente che i linguaggi esistenti non erano all'altezza del compito. Il C++ era diventato un quasi-standard per i programmatori che realizzavano applicazioni specializzate in cui la velocità era tutto... Ma il C++ non era abbastanza affidabile per quanto Gosling aveva in mente. E' veloce, ma le sue interfacce erano inconsistenti, e i programmi si bloccavano continuamente. Tuttavia, nell'elettronica

> di consumo, l'affidabilità è più importante della velocità. Le interfacce software dovevano essere affidabili come una presa elettrica. "Sono giunto alla conclusione che avevo bisogno di un nuovo linguaggio di programmazione", spiega Gosling.

Per una serie di ragioni, tra cui uno sforzo di marketing enorme da Sun Microsystems, Java è diventato sorprendentemente di successo poco tempo dopo che è stato rilasciato come un linguaggio di comunicazione di Internet a metà del 1995.

I principali componenti del sistema Java sono:

- il linguaggio di programmazione Java;
- compilatori Java e sistemi di run-time (Java Virtual Machine);
- una vasta libreria, tra cui un toolkit Java per la visualizzazione grafica e altre applicazioni e applet Java di esempio.

Anche se la libreria e toolkit hanno favorito la diffusione di Java, saremo soprattutto interessati al linguaggio di programmazione, alla sua implementazione, e al modo in cui la progettazione del linguaggio e le considerazioni sull'implementazione si furono influenzate a vicenda. Gosling, che era più modesto nella vita reale rispetto a quanto la citazione precedente farebbe dedurre, ebbe questo a dire sui linguaggi che hanno influenzato Java:

> Una delle influenze più importanti sulla progettazione di Java è stato un linguaggio molto più antico chiamata Simula. È il primo linguaggio OO che abbia mai usato (su un CDC 6400!) ... [e] in cui il concetto di *classe* fu inventato.

4.2 Obiettivi del linguaggio Java

Il linguaggio di programmazione Java e il suo ambiente di esecuzione sono stati progettati con in mente i seguenti obiettivi:

Portabilità Deve essere facile trasmettere programmi attraverso la rete ed eseguirli correttamente nell'ambiente di ricezione, indipendentemente dall'hardware, dal sistema operativo o browser web utilizzato.

Affidabilità Poiché i programmi saranno eseguiti in remoto da utenti che non hanno scritto il codice, messaggi di errore o crash dovrebbero essere evitato il più possibile.

Sicurezza L'ambiente di elaborazione che riceve un programma deve essere protetto da errori del programmatore o da programmazione dannosa.

Linking dinamico I programmi sono distribuiti in porzioni, con porzioni separate caricate nell'ambiente di runtime Java quando necessario.

Esecuzione multithread Per i programmi concorrenti che devono essere eseguiti su una varietà di hardware e sistemi operativi, il linguaggio deve includere un esplicito supporto e un'interfaccia standard per la programmazione concorrente.

Tabella 4.1: Decisioni di design di Java

Caratteristica	Portabilità	Sicurezza	Semplicità	Efficienza
Interpretato	+	+		
Type safe	+	+	+/−	+/−
Oggetti come tipi base	+/−	+/−	+	−
Oggetti mediante puntatori	+		+	−
Garbage collection	+	+	+	−
Supporto della concorrenza	+	+		

Simplicity and familiarity Il linguaggio dovrebbe essere adatto al vostro progettista media sito web, in genere un programmatore C o un programmatore parzialmente familiare con C/C++

Efficienza E' importante, ma può essere secondaria rispetto alle altre considerazioni.

In generale, la ridotta enfasi sull'efficienza ha dato ai designer Java maggiore flessibilità rispetto a quella che avevano i designer C++.

4.2.1 Le decisioni di progettazione

Alcuni degli obiettivi e delle decisioni globali di design sono elencati nella tabella 4.1, in cui + indica che una decisione che contribuisce a questo obiettivo, − indica che una decisione che va contro questo obiettivo, e +/− indica che ci sono pro e contro nella decisione. Alcune celle sono lasciate vuote, ad indicare una decisione di progettazione che ha poco o nessun effetto sull'obiettivo. Possiamo notare la relativa importanza dell'efficienza nel processo di design di Java osservando i valori lungo la colonna più a destra. Ciò non significa che l'efficienza sia stata sacrificata inutilmente, ma solo che, rispetto agli altri obiettivi, l'efficienza non era l'obiettivo primario.

Interpretato Le implementazioni iniziali e più diffuse di Java si basano su bytecode interpretato. In breve, i programmi Java sono compilati verso una forma semplificata di linguaggio di livello inferiore. Questo linguaggio, chiamato Java bytecode, è la forma che viene di solito utilizzata quando i programmi Java vengono inviati attraverso la rete come parti di pagine web. Il Bytecode Java viene eseguito da un interprete chiamato Java Virtual Machine (JVM). Un vantaggio di questa architettura è che una volta che la JVM viene realizzata per un particolare hardware o sistema operativo, tutti i programmi Java possono essere eseguiti su tale piattaforma senza alcun cambiamento. Oltre alla portabilità, il bytecode interpretato facilita l'esecuzione sicura, per esempio i comandi che violano la semantica del linguaggio Java possono essere riconosciuti prima che vengano eseguiti. Un buon esempio è l'*array-bound checking*. E non è possibile dire al momento della compilazione se un programma accederà a un array fuori dai limiti. Tuttavia, la JVM esegue dei test run-time per assicurarsi che nessun programma Java acceda alla memoria in modo non corretto per un accesso a un array fuori dai suoi limiti.

Sicurezza dei tipi Ci sono tre livelli di *type safety* in Java. Il primo è il checking di codice sorgente Java in fase di compilazione. Il type checker convenzionale di Java funziona come gli altri type checker convenzionali (come in Pascal, C++, e così via), impedendo la compilazione di programmi che non sono conformi alla disciplina dei tipi di Java. Non c'è aritmetica dei puntatori, non ci sono cast non controllati fra i tipi, e il linguaggio è *garbage collected*, fornendo un maggior grado di sicurezza dei tipi rispetto a C++, per esempio. Il secondo livello di type safety è fornito dal type-checking di Java bytecode prima della loro esecuzione. Il terzo livello è costituito da controlli di tipo run-time, come ad esempio gli *array-bounds* checks descritti nella sottosezione precedente. Oltre alla sicurezza, il type-system di Java semplifica il linguaggio in qualche misura eliminando costrutti che complicherebbero la semantica del linguaggio o l'esecuzione runtime. Ci sono alcuni costi in termini di efficienza connessi con il controllo in fase di esecuzione, comunque.

Oggetti e riferimenti In Java, molte cose sono oggetti, ma non tutte le cose. In particolare, i valori di alcuni tipi di base come numeri interi, booleani e stringhe non sono oggetti. Questo è un compromesso tra semplicità ed efficienza. In particolare, se tutte le operazioni fra interi richiedessero un method-lookup dinamico, ciò rallenterebbe considerevolmente l'aritmetica intera. Una decisione di semplificazione associata gli oggetti è che a tutti gli oggetti è eseguito l'accesso tramite puntatore, e il *pointer assignment* è l'unica forma di assegnamento prevista per tutti gli oggetti. Questo semplifica programmi eliminando alcuni casi speciali, ma in alcune situazioni riduce l'efficienza del programma.

Tutti i parametri ai metodi Java sono passati per valore Quando il parametro è un tipo riferimento (inclusi tutti gli oggetti e gli array), però, è il riferimento stesso che viene copiato e passato per valore. In effetti, questo significa che i valori di tipi primitivi sono passati per valore e gli oggetti vengono passati per riferimento.

Garbage collection Come discusso precedente, la garbage collection è necessaria per la completa sicurezza dei tipi. La garbage collection semplifica anche la programmazione, eliminando la necessità di codice che determina se la memoria può essere deallocata, ma ha un costo runtime. La garbage collection di Java sfrutta la concorrenza. In particolare, il garbage collector di Java è implementato come un thread in background a bassa priorità, permettendo alla garbage collection di verificarsi nei periodi in cui non potrebbe influenzare la percezione dell'utente della velocità di esecuzione.

Linking dinamico Le classi definite e utilizzate in un programma Java possono essere caricate nella JVM in modo incrementale, appena vengono richieste dal programma in esecuzione. Questo riduce il tempo trascorso tra l'inizio della trasmissione di un programma attraverso la rete e l'inizio di esecuzione del programma, dato che il programma può cominciare l'esecuzione prima che tutto il codice relativo sia stato trasmesso. Inoltre, se un programma termina senza bisogno di alcune classi, queste classi non devono essere trasmesse o caricate nella macchina virtuale. Il collegamento dinamico non ha un grande effetto sul design del linguaggio, a parte

il fatto di richiedere interfacce chiare che possono essere usate per controllare una parte di un programma sotto le ipotesi circa il codice fornito da un'altra parte del programma.

Supporto della concorrenza Java ha un modello di concorrenza basato sui thread, che sono processi concorrenti indipendenti. Questa è una parte significativa del linguaggio, sia perché il design è sostanziale, sia per l'importanza di avere primitive per la concorrenza standardizzate in quanto parte del linguaggio Java. Chiaramente, se i programmi Java si basassero su specifici meccanismi di concorrenza del sistema operativo, non sarebbero portabili su diversi sistemi operativi.

Semplicità Anche se Java è cresciuto negli anni, e caratteristiche come *reflection* e *inner classes* possono non sembrare "semplici", il linguaggio è comunque più piccolo e più semplice nella progettazione rispetto alla maggior parte dei linguaggi di programmazione general-purpose. Un modo per vedere la relativa semplicità di Java è quello di elencare le caratteristiche di C++ che non appaiono in Java, fra cui le seguenti:

- Structures and unions: le strutture sono *sussunte* (*subsumed*) dagli oggetti, e alcuni usi delle unioni possono essere sostituiti con le classi che condividono una superclasse comune;
- Le funzioni possono essere sostituite con metodi statici;
- L'ereditarietà multipla è complessa e molti casi può essere evitata se viene utilizzato il semplice concetto di interfaccia di Java;
- `goto` non è necessaria;
- L'overloading degli operatori è complesso e ritenuti inutile; in Java si può fare l'overloading delle funzioni;
- Le coercizioni automatiche (*automatic coercions*) sono complesse e ritenute inutili;
- i puntatori sono il valore predefinito per gli oggetti e non sono necessari per gli altri tipi. Di conseguenza non è necessario alcun tipo dedicato per rappresentare i puntatori.

Alcune di queste funzioni appaiono in C++ in primo luogo a causa dell'obiettivo di compatibilità retroattiva con C. Altri sono stati omessi da Java dopo alcune discussioni, perché è stato deciso che la complessità di includerli sia preponderante rispetto alla funzionalità avrebbero fornito. Le omissioni più significative sono l'ereditarietà multipla, le conversioni automatiche, l'overloading degli operatori, e le operazioni di puntamento nelle forme in cui si trovano in C e C++.

4.3 Classi e oggetti/eredità

Java è scritto in una sintassi simile a C++ in modo che la programmazione sia più accessibile ai programmatori C e C++:

```
class Point {
  private int x;
  public int GetX() { ... }
  protected void setX(int x) { ... }
  Point(int val) { x = val; }
}
```

Come altri linguaggi basati su classi, le classi Java dichiarano i dati e le funzioni associate con tutti gli oggetti creati da tale classe. Quando viene creato un oggetto Java, lo spazio viene allocato per memorizzare i campi dati dell'oggetto e il costruttore della classe viene chiamato per inizializzare i campi dati. Come in C++, il costruttore ha lo stesso nome della classe. Per mantenere lo stile C++, la classe `Point` ha componenti pubbliche, private e protette. Anche se public, private, e protected sono keywords di Java, le specifiche di visibilità non significano esattamente la stessa cosa nei due linguaggi.

La terminologia Java è leggermente diversa da quella di Simula, Smalltalk e C++. Ecco un breve riassunto delle caratteristiche più importanti usate per discutere di Java:

- *Class* e *object* hanno essenzialmente lo stesso significato che in altri linguaggi ad oggetti basati su classi: field = data member;
- *Method*: funzione membro, membro statico: analogo al "class field" o "class method" di Smalltalk; *this*: come il this in C++ o nello stesso Smalltalk, l'identificatore this nel corpo di un metodo Java fa riferimento all'oggetto sul quale tale metodo è stato invocato;
- *Native Method*: metodo scritto in un altro linguaggio, ad esempio C
- *Package*: insieme di classi nello stesso name space.

Prenderemo in esame diverse caratteristiche di classi e oggetti in Java, tra cui campi statici e metodi, overloading, metodi finalize, metodi main, metodi toString usati per produrre una rappresentazione visualizzabile in formato stringa dell'oggetto, e la possibilità di definire metodi nativi.

Inizializzazione Java garantisce che un costruttore venga chiamato ogni volta che viene creato un oggetto. Con l'ereditarietà nascono inoltre alcune questioni interessanti, discusse nella sezione 4.5.

Campi e metodi statici Campi e metodi statici in Java sono simili a Smalltalk variabili di classe e metodi di classe. Se un campo viene dichiarato essere statico, allora vi è un campo per l'intera classe, invece di uno per oggetto. Se un metodo viene dichiarato statico, il metodo può essere chiamato senza utilizzare un oggetto della classe. In particolare, i metodi statici possono essere chiamati anche prima della creazione di qualsiasi oggetto della classe. I metodi statici possono accedere solo ai campi statici e altri metodi statici; mentre non possono fare riferimento a "this" perché non fanno parte di un oggetto specifico della classe. L'accesso a un membro statico al di fuori di una classe di solito è effettuato con il nome della classe,

come in class_name.static_method(args), piuttosto che attraverso un riferimento a un oggetto.

I campi statici possono essere inizializzati con espressioni di inizializzazione o con un blocco di inizializzazione statico. Entrambe le modalità sono illustrate nel codice seguente:

```
class ... {
  // Variabile statica con valore iniziale
  static int x = VALORE_INIZIALE;
  // Blocco di inizializzazione statico
  static {
    // Codice da eseguire una volta sola
    // al caricamento della classe
  }
}
```

Come indicato nel commento al programma, il blocco di inizializzazione statico di una classe viene eseguito una sola volta, quando la classe viene caricata. Ci sono regole specifiche che disciplinano l'ordine di inizializzazione statica, quando una classe contiene sia le espressioni di inizializzazione sia un blocco di inizializzazione statico. Ci sono anche restrizioni alla forma di blocchi di inizializzazione statica. Per esempio, un blocco statico non può sollevare un'eccezione, in quanto non è certo che il corrispondente gestore sia installato al momento del caricamento della classe.

Overloading L'Overloading in Java si basa sulla *segnatura* di un metodo, che consiste del nome del metodo, il numero di parametri, e il tipo di ciascun parametro. Se due metodi di una classe (siano essi dichiarati nella stessa classe, o entrambi ereditati, oppure uno dichiarato e uno ereditato) hanno lo stesso nome ma segnature diverse, il nome del metodo si dice "overloaded". Come in altre linguaggi, l'overloading viene risolto al momento della compilazione.

Garbage Collection e Finalize Poiché Java è garbage collected, non è necessario liberare gli oggetti in modo esplicito. Inoltre, i programmatori non hanno bisogno di preoccuparsi di dangling references creati dalla deallocazione prematura di oggetti. Tuttavia, la garbage collection recupera solo lo spazio utilizzato da un oggetto. Se un oggetto contiene l'accesso ad un altro tipo di risorsa, ad esempio un blocco su dati condivisi, allora questo deve essere liberato quando l'oggetto non è più accessibile. Per questo motivo, gli oggetti Java possono avere metodi di finalizzazione, che sono chiamati in due circostanze: dal garbage collector appena prima che lo spazio venga recuperato, e dalla macchina virtuale quando termina. Una convenzione utile nei metodi di finalizzazione è quella di chiamare `super.finalize()`, come successivamente illustrato, in modo che venga eseguito l'eventuale codice di finalizzazione della superclasse:

```
class FileHandler extends ... {
  ...
  protected void finalize() {
    super.finalize();
    close(file);
  }
}
```

C'è un interessante interazione tra i metodi di finalizzazione e il meccanismo di eccezione Java. Eventuali eccezioni non gestite, sollevate durante l'esecuzione di un metodo `finalize()` vengono ignorate.

Un problema di programmazione associato ai metodi "finalize" è che il programmatore non ha il controllo esplicito su quando un metodo finalize viene chiamato. Questa decisione è lasciata al sistema run-time. Questo può creare problemi se un oggetto contiene un lock su una risorsa condivisa, per esempio, in quanto un lock non potrebbe essere liberato finché il garbage collector non determina che il programma richiede più spazio. Una soluzione è quella di mettere operazioni come liberare tutti i lock o altre risorse in un metodo che viene esplicitamente richiamato nel programma. Questo funziona bene, a condizione che tutti gli utenti della classe conoscano il nome del metodo e si ricordino di chiamare tale metodo quando l'oggetto non è più necessario.

Alcuni altri aspetti interessanti degli oggetti e delle classi Java sono i metodi main, utilizzati per avviare l'esecuzione del programma, i metodi `toString` utilizzati per produrre una rappresentazione "stampabile" di un oggetto, e la possibilità di definire metodi nativi:

- `main` Un'applicazione Java viene invocata con il nome della classe che guida l'applicazione. Tale classe deve avere un metodo `main`, che deve essere pubblico, statico, deve ritornare `void`, e deve accettare un solo argomento di tipo `String []`. Il metodo principale viene richiamato con gli argomenti del programma in una matrice di stringhe. Ogni classe con un metodo main può essere invocata direttamente come se fosse un'applicazione stand-alone, e tale caratteristica può essere utile per i test ad esempio.

- `toString`: Una classe può definire un metodo `toString`, che viene richiamato quando è necessaria una di conversione di tipo a `String`, come ad esempio per la "stampa" di un oggetto.

- Metodi nativi: un metodo nativo è uno scritto in un altro linguaggio, come ad esempio C. Portabilità e la sicurezza sono ridotte con metodi nativi: il codice nativo non può essere spedito in rete su richiesta, e i controlli incorporati nella JVM sono inefficaci perché il metodo non è interpretato dalla macchina virtuale. Le ragioni per l'utilizzo di metodi nativi sono:
  - l'efficienza del codice oggetto nativo;
  - l'accesso ai programmi di utilità o a programmi che sono già stati scritti in un altro linguaggio.

4.4 Pacchetti e Visibilità

Java ha quattro distinzioni di visibilità per campi e metodi, tre corrispondenti ai livelli di visibilità di C++ e una quarta derivante dai package (pacchetti).

I package Java sono un meccanismo di incapsulamento simile al namespace di C++, che permette di raggruppare insieme dichiarazioni collegate, con alcune dichiarazioni nascoste ad altri pacchetti. In un programma Java, ogni attributo o metodo appartiene ad una classe specifica e ogni classe è parte di un package. Una

classe può appartenere al package di default senza nome, o a qualche altro pacchetto, se specificato nel file che contiene la classe.

Le distinzioni di visibilità in Java sono:

- `public`: accessibile ovunque la classe è visibile.
- `protected`: accessibile ai metodi della classe e alle eventuali sottoclassi, nonché ad altre classi dello stesso pacchetto.
- `private`: accessibile solo nella classe stessa.
- package o *friendly/amichevole*: accessibile solo al codice nello stesso pacchetto; non visibile alle sottoclassi di altri pacchetti. I membri dichiarati senza un modificatore di accesso (access modifier) hanno una visibilità package.

Detto in altro modo, un metodo può fare riferimento ai membri privati della classe di appartenenza, i membri non privati di tutte le classi nello stesso package, i membri protetti di superclassi (tra cui superclassi in un pacchetto differente), e membri pubblici di tutte le classi di qualsiasi pacchetto visibile.

I nomi dichiarati in un altro pacchetto sono accessibili tramite "import", che importa le dichiarazioni da un altro package, oppure con i "qualified names" nella forma seguente, che indicano il pacchetto contenente il nome in modo esplicito:

```
// package  .class . method
   java.lang.String.substring()
```

4.5 Ereditarietà

Nella terminologia Java, una sottoclasse eredita dalla propria superclasse. Il meccanismo di ereditarietà di Java è sostanzialmente simile a quello di Smalltalk, C++ e di altri linguaggi a oggetti basati su classi. La sintassi associata con l'ereditarietà è simile a C++, con la parola chiave "extends", come mostrato in questo esempio di classe `ColorPoint` che estende la classe `Point`:

```
class ColorPoint extends Point {
  // Additional fields and methods
  private Color c;
  protected void setColor(Color d) { c = d; }
  public Color getColor() { return c; }
  // Define constructor
  ColorPoint(int xval, Color cval) {
    super(xval);  // Call Point constructor
    c = cval;     // Initialize ColorPoint field
  }
}
```

Method Overriding e Field Hiding Come in altri linguaggi, una classe eredita tutti i campi e i metodi della sua superclasse, ad eccezione di quando un campo o un metodo con lo stesso nome è dichiarato nella sottoclasse. Quando un nome metodo nella sottoclasse ha la stessa nome di metodo nella superclasse, la definizione

nella sottoclasse sovrascrive il metodo della superclasse con la stessa segnatura. Un metodo `override`, però, non può avere un tipo di ritorno diverso da quello della superclasse. Si può accedere al metodo `override` della superclasse tramite la parola chiave **`super`**. Per i campi, una dichiarazione di campo in una sottoclasse nasconde qualsiasi campo della superclasse con lo stesso nome. Un campo nascosto può accedere mediante l'uso di un "qualified name" (se è statico) o mediante l'uso di un'espressione che contiene un cast a un tipo della superclasse, oppure con la parola chiave **`super`**.

Costruttori Java garantisce che un costruttore viene chiamato ogni volta che viene creato un oggetto. Nel compilare il costruttore di una sottoclasse, il compilatore controlla che il costruttore della superclasse sia chiamato. Ciò viene fatto in un modo che i programmatori generalmente vogliono prendere in considerazione. In particolare, se la prima istruzione di un costruttore della sottoclasse non è una chiamata a **`super`**, allora la chiamata a **`super`** viene inserita automaticamente dal compilatore. Questo non sempre funziona bene, perché, se la superclasse non dispone di un costruttore senza argomenti, il **`super`** non corrisponderà un costruttore dichiarato, e vi sarà un errore in compilazione. Un'eccezione a questo controllo si verifica se un costruttore invoca un altro costruttore. In questo caso, il primo costruttore può non chiamare il costruttore della superclasse, ma il secondo costruttore deve. Ad esempio, se la dichiarazione del costruttore `ColorPoint(){ ColorPoint(0, blu); }` viene aggiunta alla classe `ColorPoint` precedente, allora questo costruttore viene compilato senza inserire una chiamata al costruttore della superclasse `Point`.

Una leggera stranezza di Java è che le convenzioni di successione per "finalize" sono diverse dalle convenzioni per i costruttori. Anche se è necessaria una chiamata alla superclasse per i costruttori, il compilatore non obbliga a una chiamata del metodo "finalize" della superclasse in un metodo "finalize" della sottoclasse.

Metodi finali e classi Java contiene un meccanismo interessante per limitare le sottoclassi di una classe: un metodo o un'intera classe possono essere dichiarate **`final`**. Se un metodo viene dichiarato **`final`**, allora il metodo non può essere sostituito (overridden) in alcuna sottoclasse. Se una classe è dichiarata **`final`**, essa non può avere sottoclassi. La ragione di questa caratteristica è che un programmatore potrebbe voler definire il comportamento di tutti gli oggetti di un certo tipo. Poiché sottoclassi producono sottotipi, questo richiede alcune limitazioni sulle sottoclassi. Per fare un esempio estremo, il pattern *Singleton* mostra come progettare una classe in modo che solo un oggetto della classe può essere creato. Il modello nasconde il costruttore della classe e rende pubblica solo una funzione che chiamerà il costruttore una volta durante l'esecuzione del programma. Questo modello risolve il problema di limitare il numero di oggetti della classe, ma solo se non esista alcuna sottoclasse che sovrascriva il metodo pubblico con un metodo che può creare più di un oggetto. Se un programmatore vuole davvero rispettare il pattern *Singleton*, ci deve essere un modo per impedire ad altri programmatori di definire sottoclassi della classe *Singleton*.

La classe `java.lang.System` è un altro esempio di una classe **`final`**. Questa classe è final in modo che i programmatori non facciano l'override dei metodi di sistema.

In un certo senso, il `final` Java è l'opposto del `virtual` di C++: i metodi Java possono essere sovrascritti eccetto quando sono marcati come `final`, mentre le funzioni di membro C++ possono essere sovrascritte solo se sono `virtual`. L'analogia non è esatta, però, perché una member function di C++ non può essere `virtual` in una classe e non `virtual` in una classe base o derivata, perché questo violerebbe l'obbligo per cui le *vtable* di classi base e derivate devono avere lo stesso layout.

La classe Object In linea di principio, ogni classe dichiarata in un programma Java estende un'altra classe, dato che una classe senza una superclasse esplicita viene interpretato come una sottoclasse della classe `Object`. La classe `Object` è una classe che non ha superclassi. La classe `Object` contiene i seguenti metodi, che possono essere sovrascritti (overridden) nelle classi derivate:

- `getClass()`, che restituisce l'oggetto `Class` che rappresenta la classe dell'oggetto. Ciò può essere utilizzato per scoprire il nome completo di una classe, i suoi membri, la sua immediata superclasse, e le eventuali interfacce che essa implementa.
- `toString()`, che restituisce una rappresentazione di stringa di un oggetto.
- `equals(Object o)` definisce la nozione di uguaglianza fra oggetti in base al valore, non al riferimento.
- `hashCode()`, che restituisce un numero intero che può essere utilizzato per memorizzare l'oggetto in una tabella hash.
- `clone()`, utilizzato per creare un duplicato di un oggetto.
- Metodi `wait()`, `notify()` e `notifyAll()`, utilizzati nella programmazione concorrente.
- `finalize()`, che viene eseguito subito prima che un oggetto venga distrutto.

Dato che tutte le classi ereditano i metodi della classe `Object`, ogni oggetto ha questi metodi.

4.6 Classi Astratte e Interfacce

Il linguaggio Java ha un meccanismo di classi astratte che è simile al C++. Una classe astratta è una classe che non implementa tutti i suoi metodi e, pertanto, non può avere alcuna istanza. Java utilizza la parola chiave `abstract` invece della sintassi "`= 0`" di C++, come mostrato nel codice seguente:

```
abstract class Shape {
  ...
  abstract Point center();
  abstract void rotate(float degrees);
}
```

Java ha anche una forma di classe "pura astratta" chiamata interfaccia. Un'interfaccia è definita in modo simile a una classe, tranne che tutti i membri di interfaccia devono essere costanti o metodi astratti. Un'interfaccia non ha un'implementazione diretta, ma le classi possono implementare un'interfaccia. Inoltre un'interfaccia può essere dichiarata come un'estensione di un altra, fornendo una forma di eredità di interfacce.

Una ragione per cui i programmatori Java utilizzano interfacce invece di classi astratte pure quando un concetto è in via di definizione, ma non implementato, è che Java permette a una singola classe di implementare diverse interfacce, mentre una classe può avere solo una superclasse. Le seguenti interfacce e classi illustrano questa possibilità. L'interfaccia `Shape` individua alcune proprietà delle forme geometriche semplici, vale a dire, ognuno ha un punto centrale e un metodo di rotazione. `Drawable` identifica in modo simile proprietà degli oggetti che possono essere visualizzati su uno schermo. Se i cerchi sono forme geometriche che possono essere visualizzate su uno schermo, allora la classe `Circle` può essere dichiarata per implementare sia `Shape` che `Drawable`, come mostrato di seguito.

Le interfacce sono spesso utilizzate come tipo di argomento a un metodo. Ad esempio, se il sistema Windows ha un metodo per disegnare oggetti su uno schermo, allora il tipo di argomento di questo metodo potrebbe essere `Drawable`, consentendo ogni oggetto che implementa l'interfaccia `Drawable` da visualizzare:

```
interface Shape {
  Point center();  // In interfaces, visibility defaults to
     public
  void rotate(float degrees);
}

interface Drawable {
  void setColor(Color c);
  void draw();
}

class Circle implements Shape, Drawable {
  // Non eredita nessuna implementazione
  // ma deve definire i metodi di Shape e Drawable
}
```

A differenza dell'ereditarietà multipla di C++, non c'è nessun problema di *name-clashing* per le interfacce Java. Più in particolare, si supponga che le precedenti due interfacce `Shape` e `Circle` anche definiscano entrambe un metodo `size()`. Se i due metodi `size()` hanno entrambi lo stesso numero di argomenti e gli stessi tipi di argomento, allora la classe `Circle` deve implementare un metodo `size()` con questo numero di argomenti e con i tipi di argomenti indicati in entrambe le interfacce. D'altra parte, se i due metodi `size()` hanno un diverso numero di argomenti o argomenti diversi per tipo, questi vengono considerati due nomi di metodi differenti e `Circle` deve definire un'implementazione per ciascuno. Poiché il method-lookup (la ricerca del metodo) di Java utilizza il nome del metodo e il numero dei tipi di argomenti per selezionare il codice del metodo, i due metodi con lo stesso nome verranno trattati separatamente al momento del method-lookup.

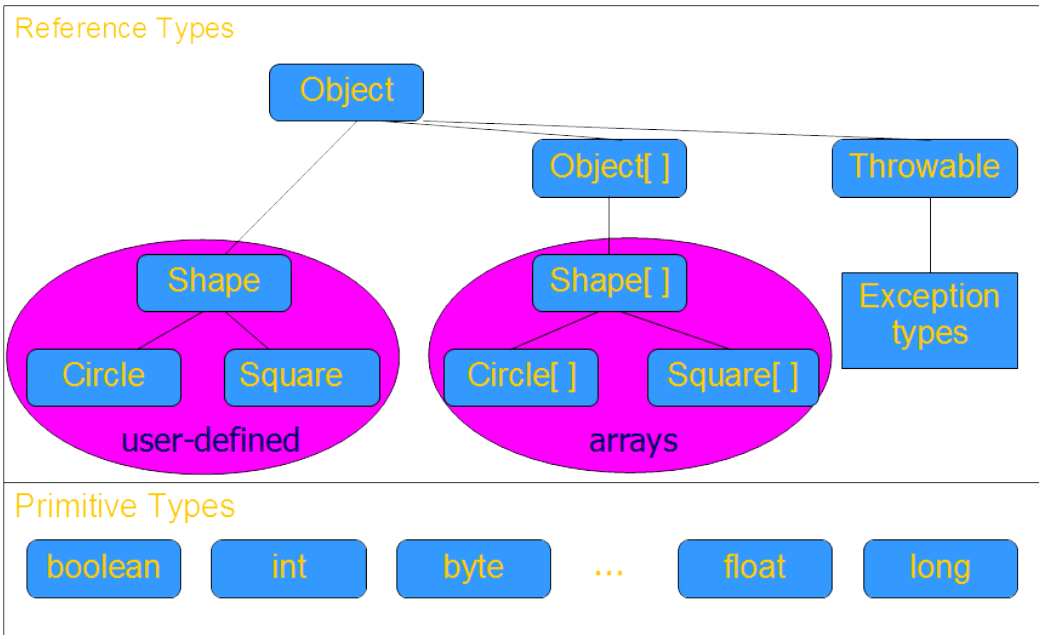


Figura 4.1: Classificazione dei tipi in Java

4.7 Classificazione dei tipi

I tipi di Java sono divisi in due categorie: i tipi primitivi (primitive types) e tipi riferimento (reference types). Gli otto tipi primitivi sono il tipo `boolean` e sette tipi numerici. I sette tipi numerici sono le forme di interi `byte`, `short`, `int`, `long`, e `char` e i tipi in virgola mobile `float` e `double`. Le tre forme di tipi riferimento sono i tipi classe, i tipi interfaccia, e tipi array. Esiste anche un tipo speciale `null`. I valori di un tipo riferimento sono riferimenti agli oggetti (che comprendono gli array). Tutti gli oggetti, compresi gli array, supportano i metodi della classe `Object`.

I rapporti sottotipizzazione tra le principali famiglie di tipi sono illustrati in fig. 4.1, che include l'interfaccia `Shape` e le classi `Circle` e `Square` per mostrare come classi e interfacce definite dall'utente rientrano nell'inquadramento appena delineato. Altri tipi predefiniti come `String`, `ClassLoader`, e `Thread` occupano posizioni simili a quella di `Exception` in questa figura. `Object[]` è il tipo di array di oggetti, e allo stesso modo per i tipi di array `Shape[]`, `Circle[]`, e `Square[]`.

Anche se chiamare Object e i suoi sottotipi come "tipi riferimento" è una terminologia standard di Java, può dare adito a confusione. Anche se C++ distingue `Object` da `Object*`, non ci sono tipi di puntatore espliciti in Java. Invece, la differenza tra un puntatore ad un oggetto e un oggetto in sé è implicita, con l'indicatore dereferenziazione combinato con operazioni come chiamata di metodo e di accesso campo. Se `T` è un tipo riferimento, allora una variabile `x` di tipo `T` è un riferimento a oggetti `T`; in C++ la variabile `x` avrebbe tipo `T*`. Poiché non esiste un modo esplicito per deferenziare `x` per ottenere un valore di tipo `T`, Java non ha un tipo separato per gli oggetti che non sono riferiti da un puntatore.

Dal momento che ogni classe è un sottotipo di `Object`, variabili di tipo `Object` possono fare riferimento a oggetti o array di qualsiasi tipo.

4.8 Subtyping per classi e interfacce

La sottotipazione (subtyping) per le classi è determinata dalla gerarchia delle classi e dal meccanismo delle interfacce. In particolare, se una classe `A` estende classe `B`, allora il tipo degli oggetti di `A` è un sottotipo del tipo degli oggetti di `B`. Non c'è altro modo per una classe di definire un sottotipo di un altro, e non esistono classi base private (come in C++) per consentire l'ereditarietà senza sottotipazione.

Una classe può essere definita per implementare una o più interfacce, il che significa che qualsiasi istanza della classe implementa tutti i metodi astratti specificati nell'interfaccia. Questo `interface subtyping` (multiplo) permette agli oggetti di supporto comportamenti comuni (multipli) senza condividere alcuna implementazione comune.

Conversione di tipo a *runtime*

Java non consente conversioni (casting) non controllate. Tuttavia, gli oggetti di un supertipo possono essere convertiti a un sottotipo tramite un meccanismo che comporta un test di tipo ("type test") a *runtime*. Se si vuole fare una lista in Java, la cosa migliore è fare in modo che possa contenere oggetti di tipo `Object`. In tal modo, oggetti di qualsiasi classe possono essere inseriti nella lista, ma per estrarli dalla lista usarli in modo non banale, essi devono poi essere convertiti al loro tipo originale (o qualche supertipo). In Java, la conversione del tipo viene controllata in fase di esecuzione (run time), sollevando un'eccezione se l'oggetto non ha il tipo designato.

Implementazione Java utilizza diverse istruzioni bytecode per la `member lookup` (ricerca membro) tramite interfaccia e di `member lookup` tramite classe o sottoclasse. In un compilatore ad alte prestazioni, la ricerca per classe potrebbe essere implementata come in C++, con offset noto al momento della compilazione. Tuttavia, la ricerca tramite interfaccia non può essere implementata come in C++, perché una classe può implementare molte interfacce e le interfacce possono elencare membri in ordini differenti.

4.9 Array, Covarianza, e Controvarianza

Per qualsiasi tipo `T`, Java ha un tipo array `T[]` i cui elementi hanno tipo `T`. Anche se i tipi array sono nello stesso gruppo di classi e interfacce, non è possibile ereditare da un tipo array. Nella terminologia Java, i tipi di array sono `final`.

Gli array sono sottotipi di Object, e quindi gli array supportano tutti i metodi associati alla classe `Object`. Come gli altri tipi riferimento, una variabile array è un puntatore ad un array e può essere null. È comune creare gli array quando un riferimento array viene dichiarato:

```
Circle[] c = new Circle[ARRAY_SIZE];
```

Tuttavia, è anche possibile creare oggetti array "anonimamente", più o meno nello stesso modo in cui siamo in grado di creare altri oggetti Java. Per esempio,

```
new int[] {1, 2, 3, ..., 10}
```

è un'espressione che crea un array intero di lunghezza 10, con valori 1, 2, 3, ..., 10. Poiché a una variabile di tipo `T[]` può essere assegnato un array di qualsiasi lunghezza, la lunghezza dell'array non è parte del suo tipo statico.

Ci sono alcune complicazioni riguardo al modo in cui tipi array Java sono collocati nella gerarchia dei sottotipi. La decisione più importante è che se `A <: B` allora il type-checker di Java utilizza anche il sottotipo `A[] <: B[]`. Questo introduce un problema spesso indicato come l' "array covariance problem". Considerate le seguenti dichiarazioni di classe e di array:

```
class A { ... }
class B extends A { ... }
B[] b = new B[10];
A[] a = b;        // Ok: B <: A => B[] <: A[]
a[0] = new A();   // Permesso, ma a runtime ho un errore di
    tipo;
                  // solleva ArrayStoreException
```

In questo codice, abbiamo `B <: A` poiché la classe `B` estende la classe `A`. Il riferimento array `b` si riferisce a un array di oggetti `B`, inizialmente tutto nullo, e il riferimento array `a` si riferisce allo stesso array. La dichiarazione di `A[] a = b` è consentita dal controllore dei tipi di Java (anche se semanticamente non dovrebbe essere consentita) a causa della decisione di progettazione di Java tale per cui se `B <: A` allora `B[] <: A[]`. Il problema del permettere a `A[] a` di fare riferimento a un array di oggetti `B` è illustrato nell'ultima riga di codice. L'assegnazione `a[0] = new A()` appare perfettamente ragionevole: `a` è un array con tipo statico `A[]`, suggerendo che è accettabile assegnare un oggetto `A` in qualsiasi posizione nell'array. Tuttavia, poiché l'array `a` si riferisce in realtà a un array di oggetti `B`, questa assegnazione violerebbe il tipo di `b`. Poiché questo assegnamento causerebbe un problema di tipi, l'implementazione di Java non consente l'esecuzione di tale assegnamento. Un test a runtime determinerebbe che il valore assegnato a un array di oggetti `B` non è un oggetto `B` e verrebbe sollevata l'eccezione `ArrayStoreException` .

Anche se i progettisti di Java considerarono la covarianza di array vantaggiosa per alcuni scopi specifici (scrivere alcune routine di copia binaria), la covarianza degli array in Java porta a una certa confusione e molti test runtime. Non sembra essere una decisione di design di successo.

4.10 Gerarchia delle classi delle eccezioni

I programmi Java possono dichiarare, sollevare, e gestire le eccezioni. Le eccezioni Java possono essere il risultato di un'istruzione **`throw`** in un programma utente o il risultato di una condizione di errore rilevato dalla macchina virtuale come un tentativo di indirizzamento al di fuori dei limiti di un array. Nella terminologia Java, un'eccezione viene sollevata (thrown) nel punto in cui si è verificata ed è catturata (caught) nel punto in cui è trasferito il controllo. Come in altri linguaggi, sollevare un'eccezione fa sì che venga fermata ogni espressione, dichiarazione, o invocazione di metodo o costruttore, e istruzione di inizializzazione che sia iniziata ma non abbia completato l'esecuzione. Questo processo continua fino a che sia trovato un gestore che corrisponde alla classe del eccezione che è stata sollevata.

Un aspetto interessante del meccanismo di eccezione Java è il modo in cui è integrato nella gerarchia delle classi e dei tipi. Ogni eccezione Java è rappresentata da un'istanza della classe `Throwable` o una delle sue sottoclassi. Un vantaggio di rappresentare eccezioni come oggetti è che un oggetto eccezione può essere utilizzato per trasportare informazioni dal punto in cui si verifica un'eccezione al gestore che la cattura. Nella gestione di un'eccezione può essere utilizzata anche la sottotipazione (subtyping): un gestore intercetta un'eccezione attraverso il suo nome di classe o tramite il nome di una superclasse della classe dell'eccezione.

Il meccanismo delle eccezioni di Java è stato progettato per funzionare bene in programmi multithread (concorrenti). Quando viene generata un'eccezione, solo il thread in cui si verifica l'eccezione è interessato. L'effetto di una eccezione sul meccanismo di sincronizzazione simultanea è che i lock vengono rilasciati non appena le dichiarazioni o le invocazioni di metodi `synchronized` terminino bruscamente.

Le eccezioni Java sono catturate all'interno di un costrutto chiamato un blocco `try-finally`. Qui di seguito è mostrato un esempio di outline di tale blocco, con due gestori, ognuno identificato dalla parola chiave `catch`. Intuitivamente, un blocco try-finally cerca di eseguire una sequenza di istruzioni. Se la sequenza di istruzioni termina normalmente, allora questo è il risultato finale del blocco. Tuttavia, se viene sollevata un'eccezione, può essere catturata all'interno del blocco. Se un'eccezione viene sollevata e catturata, la sequenza di istruzioni che seguono la parola "finally" verrà eseguita dopo che il gestore di eccezioni sia terminato:

```
try {
  ...
} catch (ExceptionType1 e1) {
  ...
} catch (ExceptionType2 e2) {
  ...
} finally {
  ...
}
```

Anche se il motivo può non essere evidente, c'è una certa complicazione nella JVM riguardo i blocchi try-finally. In particolare, una parte significativa della complessità del bytecode verificatore Java è un risultato del fatto che le clausole "finally" sono implementate come una forma di "sottoprogramma locale" (chiamato JSR) nel bytecode interprete Java.

Le classi delle eccezioni sono mostrate in fig. 4.2. Ogni eccezione è, per definizione, un oggetto di qualche sottoclasse di `Throwable`. La classe `Throwable` è una sottoclasse diretta di `Object`. Nei programmi si possono utilizzare le classi di eccezione preesistenti nelle istruzioni `throw`, oppure definire classi di eccezioni supplementari, che devono essere sottoclassi di `Throwable` o una delle sue sottoclassi. Per trarre vantaggio dal controllo dei gestori di eccezione a livello di compilazione offerto dalla piattaforma Java, è tipico definire la maggior parte delle nuove classi di eccezione come "eccezioni controllate" (checked exceptions). Queste sono sottoclassi di `Exception` che non sono sottoclassi di `RuntimeException`.

Le espressioni "eccezioni controllate" ed "eccezioni non controllate" (checked and unchecked exceptions) si riferiscono alla verifica a tempo di compilazione dell'insieme di eccezioni che possono essere sollevate in un programma Java. Il compilatore

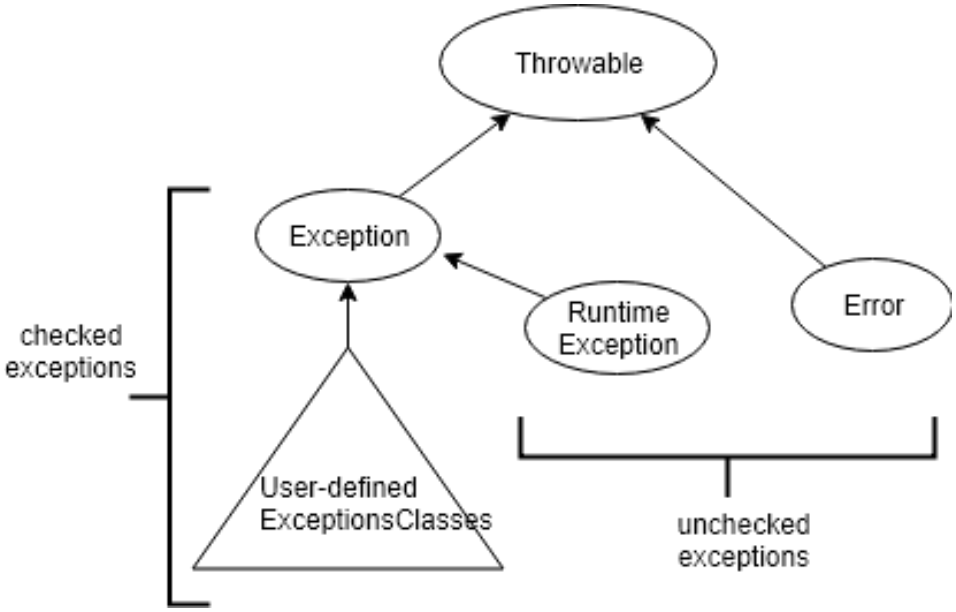


Figura 4.2: Classi di eccezioni in Java

Java controlla, in fase di compilazione, che il programma contiene i gestori per ogni eccezione controllata. Questo è realizzato analizzando le eccezioni controllate che possono essere lanciate durante l'esecuzione di un metodo o di costruzione. Tali controlli sono basati sull'insieme dichiarato delle eccezioni che un metodo Java potrebbe sollevare, che viene anche chiamato "clausola `throws`" del metodo. Per ogni eccezione controllata che è un possibile risultato di una chiamata a metodo, la clausola `throws` per tale metodo deve menzionare la classe di tale eccezione o una delle sue superclassi. Questo controllo compile-time della presenza di gestori di eccezioni è stato progettato per ridurre il numero di eccezioni che non sono adeguatamente trattate.

Dal momento che `Error` e `RuntimeException` sono eccezioni generalmente sollevate dal sistema di runtime Java e non dal codice utente, non è necessario per un programmatore dichiararle nella clausola `throws` di un metodo. Più in particolare, le classi di eccezioni di run-time (`RuntimeException` e le sue sottoclassi) sono esenti dalla verifica a tempo di compilazione perché, a giudizio dei progettisti del linguaggio di programmazione Java, dichiarare tali eccezioni non aiuterebbe in modo significativo a stabilire la correttezza dei programmi. Molte delle operazioni e dei costrutti del linguaggio di programmazione Java possono comportare eccezioni a run-time. Le informazioni a disposizione di un compilatore e il livello di analisi sulle prestazioni del compilatore di solito non sono sufficienti per stabilire che tali eccezioni di runtime non possono accadere, anche se questo può essere ovvio per il programmatore.

I programmi ordinari di solito non fanno il recovering dalle eccezioni della classe `Error` o delle sue sottoclassi. La classe `Error` è una sottoclasse separata di `Throwable`, distinta da `Exception` nella gerarchia delle classi, per consentire ai programmi di utilizzare l'idioma

```
catch (Exception e) { ... }
```

per catturare tutte le eccezioni da cui il recupero (recovery) può essere possibile senza la cattura di errori dai quali una recovery non è in genere possibile.

4.11 Binding dinamico

4.11.1 Overloading dei metodi

Come abbiamo osservato utilizzando classi già implementate, in Java si possono definire più metodi con il medesimo nome, ma con segnature differenti (overloading).

Si consideri ad esempio il seguente metodo per il calcolo del valore assoluto di un numero double:

```
public static double valoreAssoluto (double x) {
  if (x < 0) return -x;
  return x;
}
```

Questo metodo, applicato a un valore `double`, restituisce un risultato `double`. Applicando il metodo a un valore `int`, il risultato sarà sempre `double`. Volendo disporre di un metodo per calcolare il valore assoluto di numeri `int` (con risultato di tipo `int`), possiamo utilizzare l'overloading definendo nella stessa classe:

```
public static int valoreAssoluto(int x) {
  if (x < 0) return -x;
  return x;
}
```

In base al tipo dell'argomento utilizzato nella chiamata, il compilatore riconosce quale metodo dovrà essere eseguito. Si osservi che il secondo metodo si potrebbe anche scrivere come:

```
public static int valoreAssoluto(int x) {
  return (int)valoreAssoluto((double)x);
}
```

Nell'istruzione `return` viene invocato il metodo `valoreAssoluto`; il cast nell'argomento di `x` a `double` fa in modo che venga chiamato il metodo `valoreAssoluto` definito per i `double`. Tale metodo restituisce un risultato `double` che è poi forzato a `int`.

L'overloading viene risolto in fase di compilazione. Più precisamente, in fase di compilazione viene scelta la segnatura del metodo da eseguire sulla base del tipo del riferimento utilizzato per invocare il metodo (analizzando cioè i metodi disponibili per quel riferimento) e degli argomenti indicati nella chiamata. Per effettuare questa scelta, il compilatore utilizza un algoritmo che cerca, tra le differenti segnature disponibili, quella che più sì avvicina alla chiamata. Ad esempio, per la chiamata `r.m(2)` dove `r` è un riferimento e `m` è il nome di un metodo, il compilatore cercherà quella adatta fra tutte le segnature di metodi di nome `m` disponibili per il tipo di `r`.

Supponiamo che siano disponibili una segnatura con parametro `long` e una con parametro `double`, il compilatore selezionerà quella con parametro `long`: il tipo `int` utilizzato per l'argomento è infatti più vicino al tipo `long` che al tipo `double`. Nel prossimo paragrafo esamineremo in dettaglio il problema della scelta della segnatura del metodo da eseguire, che è direttamente collegata all'overloading, e il problema della scelta del metodo da eseguire, collegata invece all'overriding.

4.11.2 Overriding, overloading e scelta del metodo da eseguire

Ricordiamo che con l'overriding si riscrive in una sottoclasse un metodo della superclasse con la stessa segnatura, mentre con l'overloading è possibile definire metodi con lo stesso nome ma con segnature differenti. Abbiamo visto che il compilatore risolve l'overloading stabilendo quale metodo si debba eseguire sulla base degli argomenti utilizzati nella chiamata. In realtà, se per uno stesso metodo c'è sia overloading sia overriding, il compilatore può stabilire solo la segnatura del metodo da eseguire (*early binding*), mentre la decisione relativa al metodo effettivo, tra quelli con la segnatura selezionata, viene rimandata all'esecuzione (*late binding*). Trattiamo ora questi aspetti dettagliatamente, precisando quanto avviene in compilazione e quanto avviene in esecuzione.

Per l'analisi che segue utilizzeremo tre riferimenti: `alfa`, `beta` e `gamma`, dichiarati come:

```
A alfa;
B beta;
C gamma;
```

dove `A`, `B` e `C` sono queste classi:

```
public class A {
  private long a;
  public void assegna(long x) { a = x; }
  public String toString() { return "a = " + a; }
}
public class B extends A {
  private int b1;
  private double b2;
  public B() { b1 = 1; b2 = 1; }
  public void assegna(int x) { b1 = x; }
  public void assegna(double x) { b2 = x; }
  public String toString() {
    return super.toString()
      + ", b1 = " + b1 + ", b2 = " + b2;
  }
}
public class C extends B {
  private int c1;
  private double c2;
  public void assegna(int x) { c1 = x; }
  public void assegna(double x) { c2 = x; }
  public String toString() {
    return super.toString()
      + ", c1 = " + c1 + ", c2 = " + c2;
  }
}
```

La gerarchia di queste classi è riassunta in fig. 4.3.

Fase di compilazione: scelta della segnatura

Analizzando l'invocazione di un metodo abbiamo già evidenziato come il compilatore controlli che esista almeno un metodo invocabile per il tipo corrispondente al

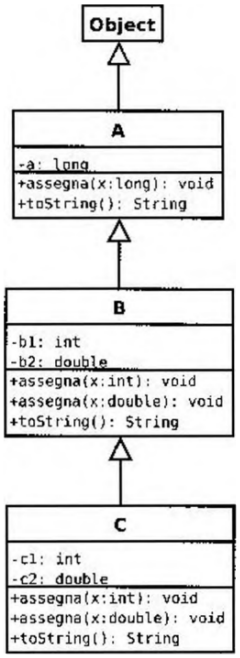


Figura 4.3: Gerarchia delle classi usate nell'esempio.

riferimento.

Ad esempio le chiamate `gamma.toString()` e `gamma.equals(alfa)` sono lecite in quanto il tipo di `gamma`, cioè la classe `C`, dispone di un metodo `toString` (definito localmente) senza parametri e di un metodo `equals` (ereditato da `Object`) con parametro di tipo `Object` (a cui verrà convertito l'argomento `alfa` di tipo `A`). A causa dell'overloading, in talune situazioni potrebbero esserci più metodi utilizzabili, o meglio potrebbero esserci differenti segnature (ad esempio se invochiamo il metodo `assegna`).

Vediamo in dettaglio le operazioni effettuate dal compilatore.

1. Scelta delle segnature "candidate" Tra le segnature di tutti i metodi con lo stesso nome di quello invocato, disponibili per la classe corrispondente al riferimento, il compilatore individua quelle che possono soddisfare la chiamata. Affinché una segnatura possa soddisfare la chiamata dev'essere:

- compatibile con gli argomenti utilizzati nella chiamata: il numero dei parametri nella segnatura e il numero degli argomenti utilizzati nella chiamata devono essere uguali; ogni argomento dev'essere di un tipo assegnabile al corrispondente parametro;
- accessibile al codice chiamante (essendo ad esempio dichiarata `public`).

Se non esistono segnature candidate, il compilatore segnala un errore. Ad esempio, nella chiamata

```
alfa.assegna(2)
```

il riferimento è di tipo `A`. L'unico metodo `assegna` disponibile per `A` è quello dichiarato in `A` stessa, che riceve un parametro di tipo `long`. Poiché l'argomento utilizzato nella chiamata può essere assegnato a un `long` (dopo una promozione), la segnatura

```
assegna(long x)
```

è una segnatura candidata (in questo caso l'unica). Per la chiamata

```
alfa.assegna(2.0)
```

l'argomento è di tipo `double`: non esistono metodi di nome `assegna` per `A` che possano ricevere argomenti di tipo `double`. La chiamata non è quindi corretta, e pertanto non viene accettata dal compilatore.

Consideriamo invece la chiamata

```
beta.assegna(2)
```

In questo caso il riferimento è di tipo `B`. Per questo tipo ci sono tre segnature candidate, e cioè:

```
assegna(int x)
assegna(double x)
assegna(long x)
```

Le prime due segnature corrispondono a metodi dichiarati direttamente nella classe `B`, la terza a un metodo ereditato da `A`.

2. Scelta della segnatura "più specifica" Tra tutte le segnature candidate, individuate nel passo precedente, il compilatore seleziona quella che più si avvicina alla chiamata, cioè quella che richiede il minor numero di promozioni. Nell'ultimo esempio, l'argomento utilizzato nella chiamata è di tipo `int`. La prima delle segnature selezionate è quindi la più adatta a questo tipo (non richiede alcuna promozione). Qualora il compilatore non riesca a individuare una segnatura più specifica delle altre, fornirà un messaggio d'errore. Un esempio di questa situazione si ha considerando una chiamata come (per semplicità omettiamo di indicare il riferimento):

```
z(1, 2)
```

nel caso siano disponibili solo due metodi z con le seguenti segnature:

```
z(double x, int y)
z(int x, double y)
```

Fase di esecuzione: scelta del metodo

Il metodo che dev'essere effettivamente eseguito viene selezionato al momento dell'esecuzione in base al tipo dell'oggetto. In particolare viene eseguilo un metodo la cui segnatura sia esattamente quella selezionata in fase di compilazione. Si ricerca tale metodo risalendo la gerarchia delle classi a partire dalla classe corrispondente all'oggetto che deve eseguire il metodo.

Consideriamo, ad esempio, la seguente classe `Prova`:

```
class Prova {
  public static void main(String[] args) {
    ConsoleOutputManager out = new ConsoleOutputManager();
```

```
    C gamma = new C();
    A alfa = gamma;
    B beta = gamma;
    out.println(alfa.toString());
    alfa.assegna(3L);
    out.println(alfa.toString());
    alfa.assegna(2);
    out.println(alfa.toString());
    beta.assegna(4);
    out.println(alfa.toString());
    gamma.assegna(8d);
    out.println(alfa.toString());
  }
}
```

Per ognuna delle chiamate, secondo quanto abbiamo spiegato sopra, il compilatore stabilisce la segnatura del metodo che dovrà essere eseguito.

Ecco il codice della classe riscritto annotando accanto a ogni chiamata di un metodo assegna un commento indicante la segnatura selezionata dal compilatore:

```
class Prova {
  public static void main(String[] args) {
    ConsoleOutputManager out = new ConsoleOutputManager();
    C gamma = new C();
    A alfa = gamma;
    B beta = gamma;
    out.println(alfa.toString());
    alfa.assegna(3L);                // assegna(long x)
    out.println(alfa.toString());
    alfa.assegna(2);                 // assegna(long x)
    out.println(alfa.toString());
    beta.assegna(4);                 // assegna(int x)
    out.println(alfa.toString());
    gamma.assegna(8d);               // assegna(double x)
    out.println(alfa.toString());
  }
}
```

Esaminiamo ora ciò che avviene in fase di esecuzione. Nella prima istruzione del metodo `main` c'è una chiamata al costruttore di `C`. Osserviamo che la classe `C` non contiene esplicitamente un costruttore. È pertanto disponibile il costruttore privo di argomenti che richiama implicitamente il costruttore privo dì argomenti della superclasse `B`. Tale costruttore è invece scritto esplicitamente. Come prima operazione, esso richiama (implicitamente) il costruttore senza argomenti della superclasse `A`. Il costruttore di `A` (anche in questo caso non esplicito) costruisce un oggetto contenente il campo `long a`, inizializzalo con il valore di default `0L`. Una volta costruito l'oggetto, il costruttore di `B` lo adatta aggiungendovi i campi `b1` e `b2` inizializzati a `1`. Infine il costruttore di `C` vi aggiunge i campi `c1` e `c2` inizializzati a `0`. Inoltre, l'oggetto verrà marcato come istanza della classe `C`.

Lo stato in cui si trova inizialmente l'oggetto può essere dunque descritto, evidenziando anche i campi ereditati dalle superclassi, come segue:

```
Istanza di C:
```

```
  Istanza di B:
    Istanza di A:
      a: 0L
    b1: 1
    b2: 1.0
  c1: 0
  c2: 0.0
```

Il riferimento al nuovo oggetto viene poi assegnato alla variabile `gamma` e, nelle istruzioni successive, alle variabili `alfa` e `beta`. Pertanto i tre riferimenti indicano il medesimo oggetto.

Per ottenere una stringa da visualizzare, viene successivamente richiamato il metodo `toString` tramite il riferimento `alfa`. Poiché l'oggetto riferito da `alfa` è un'istanza della classe `C`, sarà eseguito direttamente il metodo `toString` di `C`. Visualizzando il risultato si ottiene:

```
a = 0, b1 = 1, b2 = 1.0, c1 = 0, c2 = 0.0
```

La riga successiva contiene la chiamata

```
alfa.assegna(3L)
```

alla quale il compilatore ha associato la segnatura `assegna(long x)`. Perciò viene cercato un metodo che abbia esattamente tale segnatura a partire dalla classe effettiva dell'oggetto, cioè da `C`. Tale metodo non è presente né in `C` né in `B`, ma viene individuato in `A`. Quindi viene eseguito il metodo `assegna` di `A`, che modifica il valore del campo `a`. Lo stato dell'oggetto diventa pertanto:

```
Istanza di C:
  Istanza di B:
    Istanza di A:
      a: 3L
    b1: 1
    b2: 1.0
  c1: 0
  c2: 0.0
```

e l'istruzione successiva visualizza:

```
a = 3, b1 = 1, b2 = 1.0, cl = 0, c2 = 0.0
```

Nella successiva chiamata

```
alfa.assegna(2)
```

la segnatura selezionata dal compilatore è ancora `assegna(long x)`. Con lo stesso procedimento si cerca un metodo con la segnatura selezionata a partire dalla classe dell'oggetto, cioè da `C`. Tale metodo viene trovato in `A`. La sua esecuzione modifica nuovamente il campo `a`, ottenendo nell'istruzione successiva il seguente output:

```
a = 2, b1 = 1, b2 = 1 .0 , cl = 0 , c2 = 0.0
```

La chiamata seguente è

```
beta.assegna(4)
```

per la quale il compilatore ha selezionato la segnatura `assegna(int x)`. Nel corso dell'esecuzione viene ricercato un metodo con la stessa segnatura a partire dalla

classe dell'oggetto, cioè da `C`. Esso viene immediatamente individuato in `C`. La sua esecuzione modifica il campo `c1`. Pertanto l'output prodotto dall'istruzione seguente è:

```
a = 2, b1 = 1, b2 = 1.0, c1 = 4, c2 = 0.0
```

Infine, per la chiamata

```
gamma.assegna(8d)
```

la segnatura selezionata dal compilatore è `assegna(double x)`. Anche in questo caso, viene direttamente individuato il metodo presente nella classe `C`, che modifica il campo `c2`. L'output prodotto dall'istruzione successiva è:

```
a = 2, b1 = 1, b2 = 1.0, c1 = 4, c2 = 8.0
```

4.11.3 Il metodo `equals`

Riconsideriamo il metodo `equals` della classe `Frazione` alla luce di quanto abbiamo appena visto. A tale scopo presentiamo una classe di prova che legge i dati relativi a due frazioni e controlla se esse rappresentano lo stesso valore numerico:

```
class Equals {
  public static void main(String[] args) {
    ConsoleInputManager in = new ConsoleInputManager();
    ConsoleOutputManager out = new ConsoleOutputManager();
    int n, d;
    out.println("Inserire i dati della prima frazione:");
    n = in.readInt("Numeratore? ");
    d = in.readInt("Denominatore? ");
    Frazione f1 = new Frazione(n, d);
    out.println("Inserire i dati della seconda frazione:");
    n = in.readInt("Numeratore? ");
    d = in.readInt("Denominatore? ");
    Frazione f2 = new Frazione(n, d);
    if (f1.equals(f2))
      out.println("Le due frazioni sono uguali");
    else
      out.printIn("Le due frazioni sono diverse");
  }
}
```

Ecco un esempio di esecuzione in cui vengono fornite due frazioni con lo stesso valore:

```
Inserire i dati della prima frazione:
Numeratore? 3
Denominatore? 4
Inserire i dati della seconda frazione:
Numeratore? 6
Denominatore? 8
Le due frazioni sono uguali
```

Segue un esempio in cui vengono inserite due frazioni con valori differenti:

```
Inserire i dati della prima frazione:
Numeratore? 1
Denominatore? 4
Inserire i dati della seconda frazione:
Numeratore? 3
Denominatore? 8
Le due frazioni sono diverse
```

Riportiamo il metodo `equals` definito nell'ultima versione della classe `Frazione`. In tale versione, il costruttore memorizza la frazione in forma semplificata:

```
public boolean equals(Frazione f) {
  return this.num == f.num && this.den == f.den;
}
```

Non avendo indicato nulla nella sua intestazione, la classe `Frazione` è una sottoclasse diretta delia classe `Object`. Ricordiamo che anche `Object` ha un proprio metodo `equals` che ricevendo un parametro `Object`, restituisce `true` solo nel caso in cui l'oggetto che esegue il metodo coincida con l'oggetto di cui viene fornito il riferimento tramite il parametro. Il metodo `equals` di `Object` potrebbe dunque essere scritto come:

```
public boolean equals(Object o) {
  return this == o;
}
```

Osserviamo che i due metodi `equals` hanno segnatura differente, quindi c'è overloading e non overriding. In sostanza, la classe `Frazione` dispone di due metodi di nome equals:

- `public boolean equals(Frazione f)` è il metodo `equals` definito nella classe `Frazione`;
- `public boolean equals(Object o)` è il metodo `equals` che la classe `Frazione` eredita dalla superclasse `Object`.

Nel metodo `main` della classe `Equals`, per verificare l'uguaglianza delle frazioni all'interno dell'`if` si utilizza la condizione `f1.equals(f2)`.

Il riferimento `f1` all'oggetto che deve eseguire il metodo è di tipo `Frazione`: per esso, come abbiamo appena osservato, esistono due metodi `equals`. Poiché l'argomento `f2` è di tipo `Frazione`, il compilatore risolve l'overloading selezionando la segnatura del metodo `equals` definito nella classe `Frazione`, cioè quella con parametro `Frazione`. In questo caso, la chiamata `f1.equals(f2)` non presenta polimorfismo perché c'è un unico metodo `equals` con parametro di tipo `Frazione`.

Esaminiamo ora che cosa succede se dichiariamo ambedue i riferimenti `f1` e `f2` di tipo `Object`:

```
...
Object f1 = new Frazione(n, d);
...
Object f2 = new Frazione(n, d);
if (f1.equals(f2))
  out.println("Le due frazioni sono uguali");
else
```

```
  out.println("Le due frazioni sono diverse");
...
```

Mandando in esecuzione il codice della classe e fornendo in ingresso due frazioni con il medesimo valore, la risposta sarà che le due frazioni sono diverse:

```
Inserire i dati della prima frazione
Numeratore? 3
Denominatore? 4
Inserire i dati della seconda frazione
Numeratore? 6
Denominatore? 8
Le due frazioni sono diverse
```

Vediamo di capire la ragione di questo comportamento, a prima vista sorprendente. Analizzando l'istruzione `f1.equals(f2)` il compilatore esamina le segnature dei metodi `equals` disponibili per la classe del riferimento `f1`, che in questo caso è `Object` (non `Frazione` come nel caso precedente). L'unico metodo `equals` disponibile parametro di tipo `Object`. Dunque il compilatore seleziona la segnatura con parametro di tipo `Object`.

Come spiegato nel paragrafo precedente, durante l'esecuzione la Java Virtual Machine deve ricercare un metodo, con la segnatura stabilita dal compilatore, risalendo nella gerarchia delle classi a partire dalla classe effettiva dell'oggetto che esegue il metodo. In questo caso l'oggetto riferito da `f1` è di tipo `Frazione`. Nella classe `Frazione` non è definito un metodo `equals` con parametro `Object`. Pertanto la ricerca prosegue nella superclasse `Object`, dove invece il metodo con la segnatura cercata è presente. Tale metodo, come ricordato sopra, restituisce `true` solo se l'oggetto che lo esegue coincide con quello passato tramite il parametro. Perciò, in questa versione dell'applicazione `Equals`, il metodo `main` risponderà sempre che le frazioni sono diverse, in quanto gli oggetti cui si riferiscono `f1` e `f2` sono sempre distinti (perché creati con due `new` diverse) anche se potrebbero rappresentare lo stesso valore.

La medesima situazione si verifica se definiamo uno dei due riferimenti di tipo `Object` e l'altro di tipo `Frazione`. In particolare:

- Definendo `f1` di tipo `Object` e `f2` di tipo `Frazione` il compilatore esamina le segnature dei metodi `equals` disponibili nella classe `Object` e trova come unico metodo quello che riceve il parametro `Object`. Di conseguenza durante l'esecuzione viene selezionato come prima il metodo ereditato da `Object`. Al momento della chiamata il riferimento `f2` sarà promosso al tipo `Object`.

- Definendo `f1` di tipo `Frazione` e `f2` di tipo `Object`, il compilatore esamina le segnature dei metodi `equals` disponibili nella classe `Frazione`: quello definito in `Frazione`, con parametro `Frazione`, e quello ereditato da `Object`, con parametro `Object`. In base al tipo dell'argomento viene anche in questo caso selezionato il metodo definito in `Object`.

Nelle situazioni che abbiamo appena analizzalo c'è sempre overloading e non overriding: il metodo `equals` definito nella classe `Frazione` ha una segnatura differente dal metodo `equals` definito in `Object`. Pertanto il metodo `equals` di `Frazione` non ridefinisce quello di `Object`.

C'è overriding, e dunque si sfrutta il polimorfismo solo quando un metodo di una sottoclasse ridefinisce un metodo della superclasse utilizzando la stessa segnatura.

In realtà, quando abbiamo scritto il metodo `equals` per la classe `Frazione`, la nostra intenzione era quella di "specializzare" il metodo `equals` di `Object` alle frazioni. In altre parole, avremmo voluto ridefinire il metodo `equals` di `Object` in modo tale che, nei caso di istanze di `Frazione`, fosse in grado di verificare l'uguaglianza di due frazioni. Per ridefinire un metodo è sempre necessario utilizzare la stessa segnatura del metodo che si vuole ridefinire. Pertanto nella classe `Frazione`, al posto del metodo `equals` che abbiamo introdotto, avremmo dovuto scriverne uno con la seguente segnatura:

```
public boolean equals(Object o)
```

D'altra parte in questa intestazione il parametro è di tipo `Object`. Di conseguenza il compilatore non permette di accedere ai campi `num` e `den` (infatti non è detto che una qualsiasi istanza di `Object` possieda tali campi). In sostanza, bisogna riscrivere il codice del metodo in modo che sia in grado di trattare un qualunque parametro `Object`.

Ricordiamo che il metodo che vogliamo scrivere appartiene alla classe `Frazione` e perciò sarà eseguito da un oggetto della classe, che deve verificare l'uguaglianza con un oggetto passato come parametro. L'uguaglianza può essere decisa utilizzando per `equals` la seguente implementazione:

```
public boolean equals(Object o) {
  if (o instanceof Frazione) {
    Frazione f = (Frazione) o;
    return this.num == f.num && this.den == f.den;
  }
  return false;
}
```

Sostituendo ora nel testo della classe `Frazione` il metodo `equals` dato precedentemente con questo nuovo metodo, si ridefinisce il metodo `equals` di `Object`. Nel metodo `main` della versione della classe `Equals`, con `f1` dichiarato di tipo `Object`, a questo punto c'è polimorfismo.

Al momento della compilazione viene selezionata la segnatura di un metodo `equals` con parametro `Object` (l'unica disponibile). Al momento dell'esecuzione, poiché l'oggetto riferito da `f1` è stato costruito dalla classe `Frazione`, il metodo `equals` con parametro `Object` da eseguire sarà subito trovalo nella classe `Frazione` (si tratta appunto del metodo definito qui sopra).

In alternativa, una soluzione equivalente, ma più elegante, consiste nello scrivere nella classe `Frazione` due metodi `equals`: uno con parametro `Object` e l'altro con parametro `Frazione`. Quest'ultimo è in grado di confrontare solo due frazioni. Il primo controlla invece il tipo dell'oggetto fornito tramite il parametro; se esso è una frazione, delega il proprio compito all'altro metodo `equals`:

```
public boolean equals(Object o) {
  if (o instanceof Frazione)
    return equals((Frazione) o);
  return false;
}
public boolean equals(Frazione f) {
```

```
  return this.num == f.num && this.den == f.den;
}
```

4.11.4 Covarianza

`T` si dice covariante (rispetto alla sottotipazione di Java) se ogni volta che `A` è sottotipo di `B` allora anche `T` di `A` è sottotipo di `T` di `B` (`T` potrebbe essere il tipo ritornato da un metodo di `A`).

In Java (dalla versione 5 in poi) i valori ritornati da un metodo ridefinito possono essere covarianti. I tipi dei paramenti devono invece essere esattamente gli stessi per i metodi overriding. Altrimenti il metodo è overloaded.

Caratteristiche del tipo array:

- definito automaticamente;
- esistono per ogni classe;
- sono di tipo `final`;
- esistono array di array;
- sono riferimenti;
- può essere `null`;
- è un sottotipo di `Object[]`;
- è covariante: se `S <: T` allora `S[] <: T[]`.

4.11.5 Proprietà delle interfacce

In Java le interfacce permettono flessibilità della definizione dei tipi, evitando i problemi che derivano dall'utilizzare l'ereditarietà multipla.

Ha come svantaggio l'introduzione di un *overhead*: l'offset della tabella di ricerca non è noto in compilazione, per cui viene eseguito *bytecode* diverso in base al metodo da cercare.

4.11.6 Tipi enumerativi

Scrivere codice come

```
public static final int SEASON_WINTER = 0;
public static final int SEASON_SPRING = 1;
public static final int SEASON_SUMMER = 2;
public static final int SEASON_FALL   = 3;
```

non è typesafe. Stampare il valore non dà informazione. Per cui si è introdotto il tipo enumerativo:

```
public enum Season { WINTER, SPRING, SUMMER, FALL; }
```

Caratteristiche:

- `Comparable`;
- `toString` stampa il nome del simbolo.

4.12 Eccezioni

Funzionamento base simile a ML, C++ (costrutti `throw`, `catch`).

Alcune differenze:

- un'eccezione è un oggetto della classe `Exception`;
- usa sottotipi da abbinare a tipi di eccezioni e passarli;
- tipi dei metodi includono eccezioni che possono essere lanciate (clausola `throws` fa parte della segnatura);
- se un metodo può lanciare una eccezione controllata, questo deve essere nella clausola `throws` del metodo.

Perché definire nuovi tipi di eccezioni? Per poter passare altri dati dichiarando campi o metodi addizionali.

4.13 Varargs

I Variable Arguments (Varargs) permettono a un metodo di accettare zero o più argomenti. Prima dei varargs, era possibile fare l'overloading oppure usare un array come parametro del metodo, ma entrambe le scelte non sono considerate buone perché possono causare problemi di manutenzione. Se non si sanno quanti argomenti passare a un metodo, usare i varargs è l'approccio migliore.

```
print(String... s) {  // s é di tipo String[]
  ...
}

print("Pippo");
print("Pippo", "Pluto");
print(new String[]{"a", "b", "c"});
```

Affinché il programma compili correttamente, occorre rispettare le seguenti due regole nell'uso dei varargs:

- Ci può essere al massimo un varargs fra i parametri di un metodo.
- Se c'è il varargs fra i parametri di un metodo, deve essere l'ultimo parametro.

4.14 Visibilità in Java

Esistono 4 tipi di visibilità: `public`, `private`, `protected` e *package*.

In un determinato punto del codice, i metodi visibili sono:

- membri `private` nella stessa classe;
- membri non `private` di tutte le classi dello stesso *package*;
- membri `protected` di superclassi (anche di *package* diversi);
- membri `public` di classi nei *package* visibili.

Se ridefinisco un metodo `private` faccio overloading.

Quando ridefinisco un metodo, la visibilità può solo aumentare.

Se ridefinisco un metodo che dichiara di sollevare eccezioni, il metodo ridefinito non può sollevare più tipi di eccezioni rispetto all'originale.

4.15 Programmazione generica in Java

I generici sono stati introdotti in Java 1.5 a supporto della programmazione generica. In poche parole, i generici consentono *tipi* (classi e interfacce) come parametri durante la definizione di classi, interfacce e metodi. Proprio come i *parametri formali* più familiari utilizzati nelle dichiarazioni dei metodi, i parametri di tipo forniscono un modo per riutilizzare lo stesso codice con ingressi diversi. La differenza è che gli ingressi a parametri formali sono valori, mentre gli ingressi di tipo parametri sono tipi.

Anche prima dell'introduzione dei generici, Java supportava la programmazione generica utilizzando la classe `Object` (se scrivo un algoritmo per `Object`, questo andrà bene per qualsiasi classe).

Il codice che utilizza i generici ha molti vantaggi rispetto al codice non generico:

- Controlli di tipo più forti in fase di compilazione.

  Un compilatore Java applica un forte controllo dei tipi per gli errori di codice, che può rilevare per i tipi generici. La correzione degli errori di compilazione è più facile che correggere gli errori di runtime, che possono essere difficili da trovare.

- Eliminazione dei cast.

  Il seguente frammento di codice senza generici richiede casting:

  ```
  List list = new ArrayList();
  list.add("ciao");
  String s = (String) list.get(0);
  ```

  Quando riscritto con i generici, il codice non richiede casting espliciti:

  ```
  List<String> list = new ArrayList<String>();
  list.add("ciao");
  String s = list.get(0);
  ```

- L'attivazione di programmatori per implementare algoritmi generici.

  Utilizzando generici, i programmatori possono implementare algoritmi generici che lavorano su collezioni di diversi tipi, che possono essere personalizzato, e i tipi sono sicuri e più facili da leggere.

Come definire tipi generici

Un *tipo generico* è una classe o interfaccia generica che viene parametrizzata sui tipi.

Iniziamo esaminando una classe `Box` non generica che opera su oggetti di qualsiasi tipo. Ha solo bisogno di fornire due metodi: `set`, che aggiunge un oggetto alla casella, e `get`, che lo recupera:

```
public class Box {
  private Object object;
  public void set(Object object) { this.object = object; }
  public Object get() { return object; }
}
```

Dal momento che i suoi metodi accettano o restituiscono un `Object`, siete liberi di passare quello che volete, a patto che non sia uno dei tipi primitivi. Non c'è modo di verificare, al momento della compilazione, come viene utilizzata la classe. Una parte del codice può passare un `Integer` nel Box aspettandosi di ottenere `Integer`s in output da esso, mentre un'altra parte del codice può passare per errore una `String`, causando un errore di runtime.

Una versione generica della classe `Box`

Una *classe generica* è definita con il seguente formato:

```
class ClassName<T1, T2, ..., Tn> { ... }
```

La sezione dei parametri di tipo, delimitati da parentesi angolari (`<>`), segue il nome della classe. Essa specifica i *parametri di tipo* (chiamati anche *variabili di tipo*) `T1`, `T2`, ... e `Tn`.

Per aggiornare la classe `Box` in modo da utilizzare i tipi generici, si crea una *dichiarazione* di *tipo generico* modificando il codice da `public class Box` a `public class Box<T>`. Ciò introduce la variabile di tipo, `T`, che può essere utilizzata ovunque all'interno della classe.

Con questa modifica, la classe `Box` diventa:

```
/**
 * Generic version of the Box class.
 * @param <T> the type of the value being boxed
 */
public class Box<T> {
  private T t;
  public void set(T t) { this.t = t; }
  public T get() { return t; }
}
```

Come si può vedere, tutte le occorrenze di `Object` sono sostituite da `T`. Una variabile di tipo può essere di qualsiasi tipo **non-primitivo** si specifica: qualsiasi tipo di classe, qualsiasi tipo di interfaccia, qualsiasi tipo di array, o anche un altra variabile di tipo.

Notazione tipica per i parametri di tipo:

- `K`: chiave;
- `N`: numero;
- `T`: tipo;
- `V`: valore;
- `S`, `U`, `V`, ecc: secondo, terzo, quarto, ... tipo.

Vedrete questi nomi utilizzati in tutte le API Java SE e nel resto di questa appunti.

Invocare un'istanza di un tipo generico

Per fare riferimento alla classe Box generica all'interno del codice, è necessario eseguire un *tipo* di *invocazione generica*, che sostituisce `T` con un certo valore concreto, come ad esempio `Integer`:

```
Box<Integer> integerBox;
```

Si può pensare a un tipo di invocazione generica come simile a un metodo di invocazione normale, ma invece di passare un argomento a un metodo, si passa un *argomento di tipo* (`Integer` in questo caso) alla classe `Box` stessa.

Parametro di tipo e Argomento di tipo Molti sviluppatori usano i termini "parametro di tipo" (*Type Parameter*) e "argomento di tipo" (*Type Argument*) in modo intercambiabile, ma non sono equivalenti. Durante la scrittura del codice, si fornisce l'*argomento di tipo* al fine di creare un parametro di tipo. Pertanto, il `T` in `Foo<T>` è un parametro di tipo e la `String` in `Foo<String>` è un argomento di tipo. Questi appunti osservano questa definizione quando si utilizzano tali termini.

Come qualsiasi altra dichiarazione di variabile, il codice mostrato al paragrafo precedente non crea un nuovo oggetto `Box`. Si dichiara semplicemente che `integerBox` terrà un riferimento a una "scatola di `Integer`", che è come `Box<Integer>` viene letto.

L'invocazione di un tipo generico è generalmente nota come *tipo parametrizzato* (*parameterized type*).

Per istanziare questa classe, si utilizza la parola chiave `new`, come al solito, ma ponendo `<Integer>` tra il nome della classe e la parentesi:

```
Box<Integer> integerBox = new Box<Integer>();
```

Il diamante

In Java SE 7 e versioni successive, è possibile sostituire gli argomenti di tipo necessari per invocare il costruttore di una classe generica con un insieme vuoto di argomenti di tipo (`<>`) fino a quando il compilatore è in grado di determinarli, o dedurli, dal contesto. La coppia di parentesi angolari, `<>`, viene informalmente chiamata *diamante*. Ad esempio, è possibile creare un'istanza di `Box<Integer>` con la seguente dichiarazione:

```
Box<Integer> integerBox = new Box<>();
```

Più parametri di tipo

Come accennato in precedenza, una classe generica può avere più parametri di tipo. Ad esempio, la classe `OrderedPair` generica, che implementa l'interfaccia generica `Pair`:

```
public interface Pair<K, V> {
  K getKey();
  V getValue();
}
public class OrderedPair<K, V> implements Pair<K, V> {
  private K key;
```

```
  private V value;
  public OrderedPair(K key, V value) {
    this.key = key;
    this.value = value;
  }
  public K getKey() { return key; }
  public V getValue() { return value; }
}
```

Le seguenti istruzioni creano due istanze della classe `OrderedPair`:

```
Pair<String, Integer> p1 =
  new OrderedPair<String, Integer>("Even", 8);
Pair<String, String> p2 =
  new OrderedPair<String, Integer>("Hello", "world");
```

Il codice, `new OrderedPair<String, Integer>` crea un'istanza di `K` come `String` e `V` come un `Integer`. Pertanto, i parametri di tipo del costruttore `OrderedPair` sono `String` e `Integer`, rispettivamente. Grazie all'*autoboxing* è valido passare un `int` dove sarebbe richiesto un oggetto di tipo `Integer`.

Siccome un compilatore Java può dedurre i tipi `K` e `V` della dichiarazione di `OrderedPair<String, Integer>`, il codice appena visto può essere ridotto utilizzando la notazione *diamante*:

```
Pair<String, Integer> p1 = new OrderedPair<>("Even", 8);
Pair<String, String> p2 = new OrderedPair<>("Hello", "world");
```

4.15.1 Metodi generici

I *metodi generici* sono metodi che introducono i propri parametri di tipo (*type parameters*). La dichiarazione è simile a quella di un tipo generico, ma il contesto del parametro di tipo è limitato al metodo dove è dichiarato. Sono ammessi metodi generici statici e non statici, nonché costruttori di classe generici.

La sintassi per un metodo generico comprende un parametro di tipo, all'interno di parentesi angolari, e appare prima del tipo di ritorno del metodo. Per i metodi generici statici, la sezione del parametro di tipo deve apparire prima del tipo di ritorno del metodo.

La classe `Util` include un metodo generico, `compare`, che confronta due oggetti `Pair`:

```
public class Util {
  public static <K, V> boolean compare(Pair<K, V> p1, Pair<K,
      V> p2) {
    return p1.getKey().equals(p2.getKey())
      && p1.getValue().equals(p2.getValue());
  }
}
```

La sintassi completa per invocare questo metodo potrebbe essere:

```
Pair<Integer, String> p1 = new Pair<>(1, "apple");
Pair<Integer, String> p2 = new Pair<>(2, "pear");
boolean same = Util.compare(p1, p2);
```

4.15.2 Bounded type parameters

Ci possono essere momenti in cui si desidera limitare i tipi che possono essere utilizzati come argomenti di tipo in un tipo parametrico. Ad esempio, un metodo che opera su numeri potrebbe voler accettare solo istanze di `Number` o delle sue sottoclassi. Questo è il motivo per cui esistono i *parametri di tipo limitati* (*bounded type parameters*).

Per dichiarare un parametro di tipo limitato, elencare il nome del *type parameter*, seguito dalla parola **`extends`** e dal suo *limite superiore* (*upper bound*), che in questo esempio è `Number`. Si noti che, in questo contesto, **`extends`** viene usato in senso generale sia per indicare l'ereditarietà delle classi, sia per l'implementazione delle interfacce (cioè in questo caso non si usa **`implements`**).

```
public class Box<T> {
  private T t;
  public void set(T t) { this.t = t; }
  public T get() { return t; }
  public <U extends Number> void inspect(U u) {
    System.out.println("T: " + t.getClass().getName());
    System.out.println("U: " + u.getClass().getName());
  }
  public static void main(String[] args) {
    Box<Integer> integerBox = new Box<>();
    integerBox.inspect("some text");  // Compile-time error
  }
}
```

Modificando il nostro metodo generico per includere questo parametro di tipo limitato, la compilazione ora fallirà, dato che la nostra invocazione di inspect comprende ancora una `String`:

```
Box.java:11: error: method inspect in class Box<T> cannot be
   applied to given types;
              ^
  required: U
  found: String
  reason: inferred type does not conform to upper bound(s)
    inferred: String
    upper bound(s): Number
  where U,T are type-variables:
    U extends Number declared in method <U>inspect(U)
    T extends Object declared in class Box
```

Oltre a limitare i tipi che si possono usare per creare un'istanza di un tipo generico, parametri di tipo limitato permettono di richiamare i metodi che rispettino i limiti:

```
public class NaturalNumber<T extends Integer> {
  private T n;
  public NaturalNumber(T n) { this.n = n; }
  public boolean isEven() {
    return n.intValue() % 2 == 0;
  }
  ...
}
```

Il metodo `isEven` invoca il metodo `intValue` definito nella classe `Integer` attraverso `n`.

4.15.3 Metodi generici e parametri di tipo delimitati

I parametri di tipo delimitati sono fondamentali per l'implementazione di algoritmi generici. Si consideri il seguente metodo che restituisce il massimo all'interno di un array:

```
public static <T> T max(T[] anArray) {
  T max = anArray[0];
  for (int i = 1; I < anArray.length; i++) {
    if (anArray[i] > max)  // Compile-time error
      max = anArray[i];
  }
  return max;
}
```

L'implementazione del metodo è semplice, ma non viene compilato perché l'operatore "maggiore di" (>) si applica solo ai tipi primitivi come `short`, `int`, `double`, `long`, `float`, `byte` e `char`. Non è possibile utilizzare l'operatore > per confrontare gli oggetti. Per risolvere il problema, utilizzare un parametro di tipo delimitato dall'interfaccia `Comparable<T>`:

```
public interface Comparable<T> {
  int compareTo(T o);
}
```

Il codice risultante sarà:

```
public static <T extends Comparable<T>> T max(T[] anArray) {
  T max = anArray[0];
  for (int i = 1; i < anArray.length; i++) {
    if (anArray[i].compareTo(max) > 0)
      max = anArray[i];
  }
  return max;
}
```

4.15.4 Generics, eredità, e sottotipi (assenza di covarianza dei sottotipi)

Come già noto, è possibile assegnare un oggetto di un tipo a un oggetto di un altro tipo purché i tipi siano compatibili. Ad esempio, è possibile assegnare un `Integer` ad un `Object`, in quanto `Object` è uno dei supertipi di `Integer`:

```
Object someObject = new Object();
Integer someInteger = new Integer(10);
someObject = someInteger;  // Ok
```

Nella terminologia orientata agli oggetti, questa è detta *relazione "is a"*. Dal momento che un `Integer` *is a* sorta di `Object`, è consentita l'assegnazione. Ma `Integer` è anche una sorta di `Number`, quindi anche il seguente codice è valido:

```
public void someMethod(Number n) { ... }
someMethod(new Integer(10));   // Ok
someMethod(new Double(10.1));  // Ok
```

Lo stesso vale anche con i generici. È possibile eseguire un tipo di invocazione generica, passando numero come argomento tipo, e ogni successiva invocazione di add sarà consentita se l'argomento è compatibile con `Number`:

```
Box<Number> box = new Box<Number>();
box.add(new Integer(10));    // Ok
box.add(new Double(10.1));   // Ok
```

Consideriamo ora il seguente metodo:

```
public void boxTest(Box<Number> n) { ... }
```

Che tipo di argomento accetta? Osservando la sua segnatura, si può vedere che accetta un singolo argomento il cui tipo è `Box<Number>`. Ma che cosa significa? È consentito di passare in `Box<Integer>` o `Box<Double>`, come ci si potrebbe aspettare? La risposta è *no*, perché `Box<Integer>` e `Box<Double>` non sono sottotipi di `Box<Number>`.

Si tratta di un malinteso comune quando si tratta di programmazione con i generici, ma è un concetto importante da imparare.

Nota Dati due tipi concreti `A` e `B` (per esempio, `Number` e `Integer`), `MyClass<A>` non ha alcuna relazione con `MyClass<B>`, a prescindere dal fatto che `A` e `B` siano correlate. Il genitore comune di `MyClass<A>` e `MyClass<B>` è `Object`.

4.15.5 Classi generiche e sottotipazione

È possibile sottotipare una classe o un'interfaccia generica, estendendola o implementandola. Il rapporto tra i parametri di tipo di una classe o di un interfaccia e i parametri di tipo di un'altra (classe o interfaccia) sono determinati dalle clausole `extends` e `implements`.

Utilizzando le classi `Collections` a titolo di esempio, `ArrayList<E>` implementa `List<E>` e `List<E>` estende `Collections<E>`. Così `ArrayList<String>` è un sottotipo di `List<String>`, che è un sottotipo di `Collections<String>`. Finché non varia il tipo di argomento, la relazione di sottotipo è conservata tra i tipi.

4.15.6 Wildcards

Nei generici, il punto interrogativo (`?`), è chiamato *wildcard* (in italiano: *carattere jolly*) e rappresenta un tipo sconosciuto. Il carattere jolly può essere utilizzato in una varietà di situazioni: come tipo di un parametro, di un campo o di una variabile locale; a volte come tipo di ritorno (anche se è una buona pratica di programmazione essere più precisi). Il jolly non viene mai usato come argomento di tipo per una chiamata di metodo generico, una creazione dell'istanza classe generica, o un supertipo.

Le sezioni seguenti descrivono i caratteri jolly in modo più dettagliato, compresi i caratteri jolly delimitati superiormente (upper bounded), jolly limitati inferiormente (lower bounded), e la *wildcard capture*, che è lo scenario nel quale il compilatore inferisce automaticamente il tipo di una wildcard, ovvero nel caso dell'*unbounded wildcard*.

4.15.7 Upper bounded wildcards

È possibile utilizzare un carattere jolly limitato superiormente per allentare le restrizioni su una variabile. Ad esempio, se si desidera scrivere un metodo che funziona su `List<Integer>`, `List<Double>` *e* `List<Number>`; è possibile utilizzare un carattere jolly limitato superiormente.

Per dichiarare un jolly superiormente delimitato, utilizzare il carattere jolly (`?`) seguito dalla parola chiave **`extends`** (e non **`implements`**: anche per le implementazioni di interfacce si usa la parola chiave **`extends`**), e dalla classe (o interfaccia) che rappresenta il suo *limite superiore*.

Per scrivere il metodo che funziona su liste di `Number` e sottotipi di `Number`, ad esempio `Integer`, `Double` e `Float`, è necessario specificare `List<? extends Number>`. Il termine `List<Number>` è più restrittivo di `List<? extends Number>` perché il primo corrisponde a un elenco di tipo solo `Number`, mentre il secondo corrisponde a un elenco di tipo `Number` o una qualsiasi delle sue sottoclassi.

Si consideri il seguente metodo `process`:

```
public static void process(List<? extends Foo> list) { ... }
```

Il jolly limitato superiormente, `<? extends Foo>`, dove `Foo` è qualsiasi tipo, corrisponde a `Foo` e qualsiasi sottotipo di `Foo`. Il metodo di `process` può accedere agli elementi della lista come tipo `Foo`:

```
public static void process(List<? extends Foo> list) {
  for (Foo elem : list) {
    ...
  }
}
```

Nella clausola **`for`**, la variabile `elem` è iterata su ogni elemento della lista. Qualsiasi metodo definito nella classe `Foo` può ora essere utilizzato su `elem`.

Il metodo `sumOfList` restituisce la somma dei numeri in un elenco:

```
public static double sumOfList(List<? extends Number> list) {
  double s = 0.0;
  for (Number n : list)
    s += n.doubleValue();
  return s;
}
```

Il codice seguente, utilizzando un elenco di oggetti `Integer`, stampa `sum = 6.0`:

```
List<Integer> li = Arrays.asList(1, 2, 3);
System.out.println( "sum = " + sumOfList(li));
```

Un elenco di valori `Double` può utilizzare lo stesso metodo `sumOfList`. Il codice seguente stampa `sum = 7.0`:

```
List<Double> ld = Arrays.asList(1.2, 2.3, 3.5);
System.out.println("sum = " + sumOfList(ld));
```

4.15.8 Unbounded wildcards

Il tipo di carattere jolly illimitato viene specificato utilizzando il carattere jolly (`?`), Per esempio, `List<?>`. Questo è chiamato un *elenco di tipo sconosciuto*. Ci sono due scenari in cui un jolly illimitato è un approccio utile:

- Se si sta scrivendo un metodo che può essere implementato utilizzando funzionalità fornita nella classe `Object`.
- Quando il codice usa metodi nella classe generica che non dipendono dal parametro type; ad esempio, `List.size` o `List.clear`. In realtà, `Class<?>` è così spesso usato perché la maggior parte dei metodi di `Class<T>` non dipendono da `T`.

Si consideri il seguente metodo, `printList`:

```
public static void printList(List<Object> list) {
  for (Object elem : list)
    System.out.println(elem + " ");
  System.out.println();
}
```

L'obiettivo di `printList` è quello di stampare un elenco di qualsiasi tipo, ma non riesce a raggiungere tale obiettivo: viene stampato solo un elenco di istanze di `Object`; essa non può stampare `List<Integer>`, `List<String>`, `List<Double>`, e così via, perché non sono sottotipi di `List<Object>`. Per scrivere un metodo `printList` generica, usare `List<?>`:

```
public static void printList(List<?> list) {
  for (Object elem: list)
    System.out.print(elem + " ");
  System.out.println();
}
```

Poiché per ogni tipo concreto `A`, `List<A>` è un sottotipo di `List<?>`, è possibile utilizzare `printList` per stampare un elenco di qualsiasi tipo:

```
List<Integer> li = Arrays.asList(1, 2, 3);
List<String> ls = Arrays.asList("uno", "due", "tre");
printList(li);
printList(ls);
```

4.15.9 Lower bounded wildcards

La sezione 4.15.7 mostra che un jolly limitato superiormente limita il tipo sconosciuto ad essere uno specifico tipo o un sottotipo di esso, ed è rappresentato utilizzando la parola chiave `extends`. In modo simile, un carattere jolly *delimitato inferiormente* limita il tipo sconosciuto ad essere un tipo specifico o un *super tipo* di quel tipo.

Un jolly limitato inferiormente è espresso utilizzando il carattere jolly (`?`), Seguito dalla parola chiave `super`, seguita dal suo *limite inferiore*: `<? super A>`.

Assumiamo che si desidera scrivere un metodo che mette degli oggetti `Integer` in una lista. Per massimizzare la flessibilità, si desidera che il metodo debba lavorare su `List<Integer>`, `List<Number>`, e `List<Object>` (tutto ciò che può contenere valori `Integer`).

Per scrivere il metodo che funziona su liste di `Integer`, oppure su liste di supertipi di `Integer`, come `Number` e `Object`, è necessario specificare `List<? super Integer>`. Il termine `List<Integer>` è più restrittiva di `List<? super Integer>` perché il primo corrisponde a un elenco di tipo solo `Integer`, mentre il secondo corrisponde a un elenco di qualsiasi tipo che è un supertipo di `Integer`.

Il codice seguente aggiunge i numeri da 1 a 10 alla fine di un elenco:

```
public static void addNumbers(List<? super Integer> list) {
  for (int i = 1; i <= 10; i++)
    list.add(i);
}
```

Come descritto prima, classi o interfacce generiche non sono legate solo perché vi è una relazione tra i loro tipi, ma è possibile utilizzare i caratteri jolly per creare una relazione tra classi generiche o interfacce.

Date le seguenti due classi regolari (non generiche):

```
class Employee { ... }
class Manager extends Employee { ... }
```

sarebbe ragionevole di scrivere il seguente codice:

```
Manager m = new Manager();
Employee e = m;
```

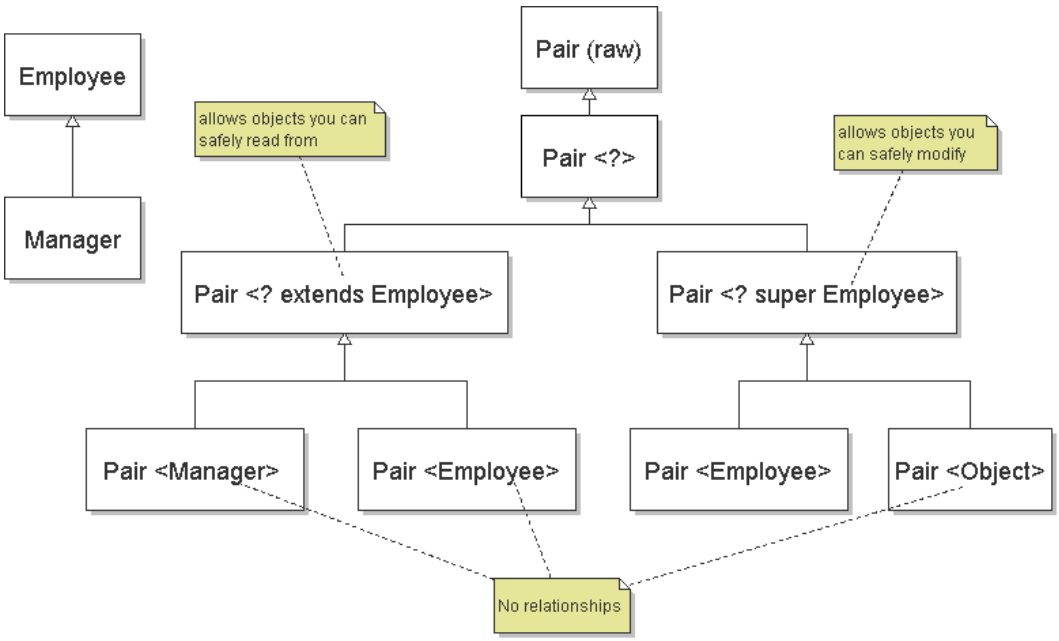
Questo esempio mostra che l'ereditarietà di classi regolari segue questa regola di sottotipo: `B` è un sottotipo di classe `A` se `B` estende `A`. Questa regola non si applica ai tipi generici:

```
List<Manager> lm = new ArrayList<>();
List<Employee> le = lm;  // Compile-time error
```

Dato che `Manager` è un sottotipo di `Employee`, qual è il rapporto tra `List<Manager>` e `List<Employee>`?

Anche se `Manager` è un sottotipo di `Employee`, `List<Manager>` non è un sottotipo di `List<Employee>` e, in effetti, questi due tipi non sono correlati. Il genitore comune di `List<Employee>` e `List<Manager>` è `List<?>`.

Esistono però anche relazioni tra i tipi intermedi come `List<? extends Employee>`, `List<? super Employee>`, e `List<Employee>`.



5 C++

5.1 Introduzione

5.1.1 Storia

Lo sviluppo del linguaggio C++ all'inizio degli anni Ottanta è dovuto a Bjarne Stroustrup che, nell'ambito della sua collaborazione con i laboratori Bell, avviò una ricerca per la definizione di un nuovo linguaggio, originariamente noto con il nome di "C with classes" (C con le classi).

Tale scelta fu maturata a seguito della sua precedente esperienza con Simula, uno dei primi linguaggi orientati alla programmazione a oggetti, del quale Stroustrup desiderava incorporare alcune caratteristiche nel linguaggio C al fine di risolvere alcune simulazioni molto rigorose, applicando il paradigma della programmazione a eventi. Per questo tipo di applicazione la scelta della massima efficienza precludeva infatti l'impiego di altri linguaggi.

Nel 1983 il nome del linguaggio fu cambiato da "C con classi" a C++. Furono aggiunte nuove funzionalità, tra cui funzioni virtuali, overloading di funzioni ed operatori, *reference*, costanti, controllo dell'utente della gestione della memoria, type checking migliorato e commenti nel nuovo stile (`//`).

Nel 1989 furono introdotte l'ereditarietà multipla, le classi astratte, le funzioni membro statiche, le funzioni membro `const`, e i membri protetti. Le ultime aggiunte di funzionalità includono i `template`, le eccezioni, i `namespace`, i nuovi tipi di cast ed il tipo di dato `bool`eano.

Le ultime revisioni C++11 e C++14 (rispettivamente del 2011 e del 2014) hanno aggiunto ulteriori funzionalità che mirano a semplificare la scrittura di codice e ad aggiungere alcune caratteristiche dei linguaggi funzionali.

5.1.2 Scopo

L'obiettivo principale del C++ è quello di essere pienamente compatibile col C ma aggiungendo il supporto per la programmazione orientata agli oggetti senza compromettere l'efficienza. Una regola fondamentale nel design del C++ è che *you don't pay for what you don't use*, cioè: se una feature non viene sfruttata in una certa porzione del codice, l'efficienza di quel codice sarà come se quella feature non esistesse nemmeno (es.: se una classe non ha metodi o classi base virtuali, allora l'occupazione di memoria di un oggetto di tale classe è pari alla somma dell'occupazione di memoria dei suoi campi).

5.2 Classi e dati astratti

5.2.1 Astrazione

Il concetto generale di astrazione è stato già spiegato nei capitoli precedenti, ora vediamo come si applica nel linguaggio C++.

In C++ si possono utilizzare `struct` simili a quelle presenti in C:

```
#include <iostream>
#include <iomanip>
using namespace std;

struct Time { int hour, minutes, seconds; };

void printMilitary(Time const&);
void printStandard(Time const&);

int main() {
  Time dinnerTime;

  dinnerTime.hour    = 18;
  dinnerTime.minutes = 30;
  dinnerTime.seconds =  0;

  printMilitary(dinnerTime);  // 18:30:00
  printStandard(dinnerTime);  // 06:30:00 PM

  dinnerTime.hour    =  29;   // Valori
  dinnerTime.minutes =  73;   // non
  dinnerTime.seconds = 103;   // validi

  printMilitary(dinnerTime);  // 29:73:103
}

void printMilitary(Time const& t) {
  cout << right << setfill('0')
       << setw(2) << t.hour    << ':'
       << setw(2) << t.minutes << ':'
       << setw(2) << t.seconds;
}

void printStandard(Time const& t) {
  cout << right << setfill('0')
       << setw(2) << (t.hour % 12 == 0) ? 12 : t.hour % 12 <<
          ':'
       << setw(2) << t.minutes << ':'
       << setw(2) << t.seconds
       << t.hour < 12 ? " AM" : " PM";
}
```

Si può notare che l'utilizzo di queste `struct` presenta alcuni svantaggi:

- una `struct` può essere utilizzata senza che i suoi campi siano inizializzati;
- si possono assegnare valori non validi ai campi;

- se l'implementazione della **struct** cambia, tutto il codice che usa quella **struct** deve essere modificato di conseguenza.

Riscriviamo ora lo stesso programma sfruttando i costrutti resi disponibili dal C++ per l'astrazione dei dati:

```
class Time {
public:
  Time();  // Constructor
  void setTime(int, int, int);
  void printMilitary();
  void printStandard();
private:
  int hour, minutes, seconds;
};
```

Il costruttore inizializza tutti i campi a 0, non ritorna alcun valore, deve chiamarsi con il nome della classe e permette di rendere tutti gli oggetti di tipo `Time` consistenti:

```
Time::Time() { hour = minutes = seconds = 0; }
```

I valori dei campi possono essere modificati solo tramite l'utilizzo del metodo `setTime`:

```
void Time::setTime(int h, int m, int s) {
  hour    = (h >= 0 && h < 24) ? h : 0;
  minutes = (m >= 0 && m < 60) ? m : 0;
  seconds = (s >= 0 && s < 60) ? s : 0;
}
```

Di seguito il resto del codice:

```
void Time::printMilitary() {
  cout << right << setfill('0')
       << setw(2) << hour    << ':'
       << setw(2) << minutes << ':'
       << setw(2) << seconds;
}

void Time::printStandard() {
  cout << right << setfill('0')
       << setw(2) << (hour % 12 == 0 ? 12 : hour % 12) << ':'
       << setw(2) << minutes << ':'
       << setw(2) << seconds
       << (hour < 12 ? " AM" : " PM");
}

int main() {
  Time t;  // Calls the constructor

  t.printMilitary();     // 00:00:00
  t.printStandard();     // 12:00:00 AM

  t.setTime(13, 27, 6);
  t.printMilitary();     // 13:27:06
  t.printStandard();     // 01:27:06 PM
```

```
  t.setTime(99,99,99);
  t.printMilitary();      // 00:00:00
  t.printStandard();      // 12:00:00 AM

  return 0;
}
```

I campi della classe sono dichiarati `private`, quindi sono accessibili solo all'interno della classe. Inserendo valori non validi, ai campi della classe vengono assegnati i valori `(0, 0, 0)`, che sono validi.

La classe appena dichiarata diventa un tipo, e quindi può essere usata per dichiarare variabili allo stesso modo dei tipi built-in come `int` e `double`:

```
Time sunset;
Time timeArray[5];
Time* timePointer;
```

Non lasciatevi ingannare dall'esempio appena visto: in C++ `struct` e `class` sono praticamente la stessa cosa; quello che cambia è solo la visibilità di default del contenuto. Infatti, le due seguenti righe di codice sono equivalenti:

```
struct T { ... };
class T { public: ... };
```

come lo sono le seguenti due:

```
class T { ... };
struct T { private: ... };
```

In genere si consiglia di usare una `class` nel caso in cui sia possibile stabilire un *invariante* sui dati contenuti, e una `struct` negli altri casi.

5.2.2 Costruttore

Il *costruttore* è il metodo utilizzato per l'inizializzazione di un oggetto non appena viene creato. Può essere *overloaded*:

```
class Time {
public:
  Time() { hour = minutes = seconds = 0; }
  Time(int h) { setTime(h, 0, 0); }
  Time(int h, int m, int s) { setTime(h, m, s); }
  void setTime(int h, int m, int s) { ... }
  ...
private:
  int hour, minutes, seconds;
};
```

Come si usa il costruttore?

```
// Time::Time()
Time t1;
Time t1 ();  // Error!

// Time::Time(int)
Time t2 (8);
Time t2 = 8;
```

```
Time t2 = Time(8);
Time t2 = (Time)8;

// Time::Time(int, int, int)
Time t3 (8, 15, 14);
Time t3 = Time(8, 15, 14);

// Pointers
Time* t;
pt = new Time;
pt = new Time();
pt = new Time(8);
pt = new Time(8, 15, 14);

// Arrays
// Default construct each element
Time at[5];
// Explicitly construct first 4 elements, default construct
   the others
Time at[6] = {3, Time(5), Time(), Time(1,12,3)};
at[0].printMilitary();  // 03:00:00
at[1].printMilitary();  // 05:00:00
at[2].printMilitary();  // 00:00:00
at[3].printMilitary();  // 01:12:03
at[4].printMilitary();  // 00:00:00
at[5].printMilitary();  // 00:00:00

// Pointers to arrays
// Default construct each element (no other option)
Time* pat = new Time[8];
Time* pat = new Time[10][20];
```

Member initializer list del costruttore

Nel definire il costruttore di una classe c'è una notazione specifica che permette di inizializzare i campi della classe (alcuni o tutti) prima dell'esecuzione del contenuto del costruttore:

```
class Info {
public:
  Info();
private:
  int const i;
  double m;
  Time t;
};
Info::Info(int j, double n) : i(j), m(n), t(i) {
  // Additional code to be executed after the
  // initialization of the fields
  ...
}
```

Copy-constructor

In C++ è possibile dichiarare un costruttore da essere utilizzato quando un oggetto di un certo tipo viene inizializzato utilizzando un altro oggetto dello stesso tipo:

```
struct S { ... };

int main() {
  S s (...);  // Calls S's constructor
  S t (...);  // Calls S's constructor
  S u (s);    // Calls S's copy-constructor
  S v = s;    // Calls S's copy-constructor
  t = v;      // Calls S's copy-assignment operator
}
```

La segnatura solita per la dichiarazione del *copy-constructor* di un tipo `S` è

```
struct S {
  S(S const&);
};
```

deve cioè accettare un parametro che è un riferimento (solitamente) `const` dello stesso tipo.

Nel caso in cui il *copy-constructor* non fosse dichiarato, esso viene generato automaticamente dal compilatore, producendo una copia *shallow* dell'oggetto (tutti i campi copiati per valore, anche i puntatori):

```
struct Box {
  int* v;
  Box(int* x) { v = x; }
  ~Box() { delete v; }
};
int main(int argc, char* argv[]) {
  int* x = new int(argc);
  Box b (x);

  if (*b.v > 4) {
    Box c = b;
    ...
  }  // Calls Box::~Box for c!
  // b.v could now be a dangling pointer
  ...
}  // Calls Box::~Box for b! x could be deleted twice
```

Per questo motivo, in genere quando si rende necessario dichiarare esplicitamente un distruttore, la stessa cosa vale per il *copy-constructor*.

5.2.3 Distruttore

Il *distruttore* è il metodo duale del *costruttore*, e viene utilizzato per garantire il rilascio delle risorse acquisite da un oggetto di una determinata classe quando questo oggetto esce dallo *scope*.

Per esempio, per avere un vettore allocato dinamicamente in C si potrebbe scrivere:

```
#include <stdlib.h>  // atoi
```

```
int main(int argc, char *argv[]) {
  if (argc < 2)
    return 1;
  int N = atoi(argv[1]);
  int *arr = (int*) malloc(N * sizeof(int));
  ...
  free(arr);
  return 0;
}
```

Tutte le volte che si utilizza la funzione `malloc`, bisogna ricordarsi di invocare (una ed una sola volta) la `free` corrispondente.

La stessa cosa vale in C++ utilizzando gli operatori **`new`** e **`delete`**, a meno di incapsulare la risorsa allocata in una classe che la gestisca automaticamente attraverso l'utilizzo della coppia *costruttore/distruttore*:

```
#include <cstdlib>  // atoi
class Vector {
public:
  Vector(int size = 10);  // Constructor
  ~Vector();              // Destructor
  ...
private:
  int* arr;
};
Vector::Vector(int size) { arr = new int[size]; }
Vector::~Vector() { delete arr; }

int main(int argc, char* argv[]) {
  if (argc < 2)
    return 1;
  int N = atoi(argv[1]);
  Vector v (N);
  ...
}  // Destructor automatically called for 'v'
```

Ogni volta che un oggetto della classe `Vector` viene inizializzato (tramite il costruttore), della memoria viene allocata; tale memoria viene automaticamente deallocata (grazie al distruttore) quando l'oggetto di tipo `Vector` esce dallo *scope*, cioè quando non ci si può più riferire a quell'oggetto.

Come dichiarare il distruttore

il distruttore va dichiarato come metodo il cui nome deve coincidere col nome della classe, preceduto da una tilde (~); il tipo ritornato non va specificato, ed inoltre non accetta parametri.

Come utilizzare il distruttore

Il distruttore viene chiamato *automaticamente* alla fine del *lifetime* dell'oggetto oppure durante la chiamata della procedura **`delete`** eseguita su un puntatore ad un oggetto di quel tipo. Quasi mai è necessario chiamare il distruttore esplicitamente.

Considerando la classe `Vector` dichiarata poco sopra:

```
int main() {
  Vector v (10);
  Vector* pv = new Vector(20);
  Vector* av = new Vector[10];
  ...
  delete pv;     // Calls Vector::~Vector() for '*pv'
  delete[] av;   // Calls Vector::~Vector() for 'arr[i]'
                 // where 0 <= i < 10
}  // Calls Vector::~Vector() for 'v'
```

Il distruttore di una classe invoca automaticamente il distruttore di tutti i campi della classe stessa; se dichiarato `virtual`, invoca anche il distruttore delle super-classi della classe considerata.

Nota sulla `delete` Invocare un'operazione di `delete` su un puntatore più di una volta è scorretto e può portare a errori del codice. Se però il puntatore è `NULL`, l'operazione di `delete` non fa nulla. Per questo spesso si raccomanda di mettere a `NULL` un puntatore subito dopo averlo cancellato, per evitare di cancellarlo due volte:

```
delete p;
p = 0;  // Equivalent to 'p = NULL'
```

E' anche utile nel caso di continui per sbaglio ad usare il puntatore dopo la `delete`, infatti, mettere a `NULL` e poi usare il puntatore genera facilmente un errore.

5.2.4 Funzioni

La dichiarazione di una funzione in C++ è simile a Java:

```
ReturnType functionName(Type1 arg1, ..., TypeN argN);
```

I metodi delle classi possono contenere la parola chiave `const` dopo l'elenco degli argomenti:

```
class ClassName {
  ...
  ReturnType methodName(...) const;
};
```

Un metodo dichiarato `const` non può modificare i membri della classe. Solo i metodi `const` possono essere invocati su un oggetto dichiarato anch'esso `const`:

```
class Car {
  int length;
  double weight;
public:
  ...
  void car_accident() { length /= 2; }
  int fun_weight(double new_weight) const {
    weight++;  // Error!
    new_weight += weight;
    return (int)new_weight;
  }
};
```

```
int main() {
  Car c;
  int w = c.fun_weight(5);  // Ok
  c.car_accident();         // Ok
  Car const cc = c;
  w = cc.fun_weight(5);     // Ok
  cc.car_accident();        // Error!
}
```

I modificatori di visibilità (`public`, `protected` e `private`) non fanno parte della dichiarazione di un metodo.

Esempi di dichiarazione di funzioni:

```
// Declaration doesn't need arguments names
void output(const std::string&);

// 'const' specifier can equivalently be before or after the
   type
double multiply(const double fac1, double const fac2);

void doSomething(SomeBigObject  o);  // Pass-by-value
void doSomething(SomeBigObject* o);  // Pass-by-pointer
void doSomething(SomeBigObject& o);  // Pass-by-reference
```

Chiamata per valore

Quando un argomento di una funzione viene passato *per valore*, la chiamata comporta la creazione di una copia del parametro nella memoria locale alla funzione. Da ciò deriva che i cambiamenti a tale variabile sono locali (quindi una volta terminata la funzione i cambiamenti non saranno più presenti).

È ampiamente inefficiente con oggetti molto grandi, poiché sono copiati interamente sullo stack.

Chiamata per riferimento

Questa chiamata comporta la memorizzazione sullo stack dell'indirizzo di memoria del parametro passato; è quindi molto efficiente in caso di oggetti di grandi dimensioni. Inoltre le modifiche apportate nella funzione sono visibili anche dalla funzione chiamante.

In C++ la chiamata per riferimento può essere realizzata in due modi quasi equivalenti:

```
void doSomething(Data* data);  // Uses pointer
void doSomething(Data& data);  // Uses reference
...
int main() {
  Data d = ...;
  Data* pd = &d;
  doSomething(&d);     // Calls 'doSomething(Data*)'
  doSomething(d);      // Calls 'doSomething(Data&)'
  doSomething(pd);     // Calls 'doSomething(Data*)'
  doSomething(*pd);    // Calls 'doSomething(Data&)'
```

```
  doSomething(NULL);  // Calls 'doSomething(Data*)'
}
```

L'utilizzo dei riferimenti presenta qualche vantaggio:

- si può utilizzare la variabile all'interno della funzione come se non fosse un puntatore;
- il riferimento passato non può essere `NULL`.

Funzioni `inline`

Una funzione `inline` è una funzione per cui la chiamata viene sostituita con il corpo della funzione (allo stesso modo di quanto succede per le macro del pre-processore). Questo permette di eliminare l'overhead associato alla chiamata di una funzione. Questo è utile soprattutto per funzioni di qualche istruzione.

Una funzione è `inline` se viene definita all'interno del corpo di una classe, oppure se viene esplicitamente definita come tale; in questo caso, la funzione dev'essere definita nel file `.h` dove è dichiarata (e non nel corrispondente file `.cpp`).[I]

```
/* IntPtr.h */
class IntPtr {
  int* p;
public:
  IntPtr(int v) {             // Decl. and def.: inline
    p = new int(v);
  }
  ~IntPtr() { delete p; }     // Decl. and def.: inline
  int& get();                 // Decl.
  int get() const;            // Decl.
};

IntPtr add(IntPtr, IntPtr);   // Decl.
IntPtr sub(IntPtr, IntPtr);   // Decl.

inline int& IntPtr::get() {   // Def.: inline
  return *p;
}

inline IntPtr
add(IntPtr a, IntPtr b) {     // Def.: inline
  return IntPtr(a.get() + b.get());
}

/* IntPtr.cpp */
int IntPtr::get() const {     // Def.: not inline
  return *p;
}

IntPtr
sub(IntPtr a, IntPtr b) {     // Def.: not inline
  return IntPtr(a.get() - b.get());
```

[I] Una funzione può essere definita all'interno di un file `.h` solo se è definita come `inline`.

```
}
```

Le funzioni ricorsive definite `inline` vengono "espanse" un certo numero di volte, solitamente impostabile al momento della compilazione.

Valori di default per gli argomenti

Quando le funzioni hanno una lunga lista di parametri, è tedioso scriverli (e difficile leggerli) durante la chiamata della stessa. Il C++ permette di specificare, nella dichiarazione di una funzione (o metodo), il valore di default da utilizzare nel caso il parametro non venga passato quando la funzione viene invocata. I valori di default, se presenti, vanno specificati partendo dall'argomento più a destra, senza saltare nessun argomento:

```
void f(int size, int initQuantity = 0);  // Ok
void g(int x, int = 0, float = 1.1);     // Ok
void h(int = 0, int x, float = 1.1);     // Error!
```

Overloading

In C++ è possibile fare l'*overloading* delle funzioni come in Java; è possibile cioè dichiarare funzioni con lo stesso nome ma diversa lista degli argomenti:

```
void f(int size, int initQuantity);
void f(int size, double initQuantity);  // Ok
void f(int size);                       // Ok
```

Non è possibile invece cambiare il tipo ritornato:

```
void f(int size, int initQuantity);
int  f(int size, int initQuantity);  // Error!
```

Il compilatore esegue la versione corretta della funzione in base al numero/tipo di argomenti della funzione chiamata.

Questa feature viene ampiamente sfruttata soprattutto per i costruttori.

5.2.5 Storage duration e linkage

In C++ esistono delle keyword (`static` ed `extern`) che permettono di modificare *storage duration* e *linkage* di variabili e funzioni.

5.2.6 Storage duration

Si riferisce al *lifetime* di una variabile. Può essere:

automatic L'oggetto viene allocato quando il programma entra nello *scope* che lo dichiara e deallocato alla fine dello stesso. Sono tali tutte le variabili locali, tranne quelle dichiarate `static` o `extern`.

static L'oggetto viene allocato all'inizio del programma e deallocato al termine. Sono tali tutte le variabili globali (anche dentro `namespace`) e quelle dichiarate `static` o `extern`.

dynamic L'oggetto viene allocato e deallocato utilizzando operazioni come `new` e `delete`.

5.2.7 Linkage

Si riferisce alla visibilità della dichiarazione (nome) di una variabile, funzione o tipo (`class`, `struct`, `typedef`, ...). Se una variabile, funzione o entità di altro tipo viene dichiarata in *scope* diversi pur non avendo un *linkage* sufficientemente ampio, allora più istanze della stessa vengono generate.

Il *linkage* può essere:

no linkage Il nome è visibile solo all'interno dello *scope* in cui viene dichiarato. Sono tali tutte le dichiarazioni locali che non sono `extern`.

internal linkage Il nome è visibile da tutti gli *scope* nell'unità di compilazione[II] corrente. Sono tali tutte le variabili e le funzioni globali dichiarate `static`, e le variabili globali `const` non dichiarate `extern`.

external linkage Il nome è visibile anche in altre unità di compilazione. Sono tali tutte le funzioni non dichiarate `static`, le variabili globali non `const` e non `static`, le variabili globali `extern`, tutti i tipi e i membri `static` delle classi.

Va precisato inoltre che:

- Le variabili locali dichiarate `static` sono inizializzate solo la prima volta che il programma entra nello *scope* in cui sono definite.
- I membri `static` di una classe non sono associati ad uno specifico oggetto di quella classe, ma piuttosto alla classe stessa. I metodi `static` non possono accedere a membri non `static` della classe, non possono usare il puntatore `this` né essere dichiarati `virtual`. Costruttori e distruttori non possono essere dichiarati `static`.

Di seguito alcuni esempi.

```
/* a.cpp */
#include <iostream>
int g;                  // static, external linkage
int const g_c = 0;      // static, internal linkage
static int g_s = 1;     // static, internal linkage
extern int g_i = 1;     // static, external linkage
// warning: 'g_i' initialized and declared 'extern'
// multiple definition of 'g_i'

static void f_s() {
  g++;
  g_s++;
}

void f() {
  static int i = 0;
  std::cout << ++i << std::endl;
  g++;
  f_s();
```

[II] Un'unità di compilazione è sostanzialmente un file `.cpp` con tutti i file `.h` che include.

```
}

/* b.cpp */
extern int g;           // static, external linkage
extern int const g_c;   // static, external linkage
extern int g_s;         // static, external linkage
int g_i;                // static, external linkage

void f();
void f_s();

int main() {
  g++;
  g_i = 4;
  f();            // Prints '1'
  f();            // Prints '2'
  f_s();          // Error: undefined reference to 'f_s()'
  int x = g_c;    // Error: undefined reference to 'g_c'
  g_s++;          // Error: undefined reference to 'g_s'
}
```

```
/* Car.h */
class Car {
  static int num_cars;
public:
  Car() { num_cars++; }
  static int n_cars();
};

/* Car.cpp */
int Car::num_cars = 0;     // Initialized from outside class
                           // also if private
int Car::n_cars() { return num_cars; }

int main() {
  int i = Car::num_cars;   // Error! 'num_cars' is private
  Car c;
  int j = Car::n_cars();   // j = 1
  Car d;
  j = d.n_cars();          // Ok, but bad style
}
```

Layout in memoria

In fig. 5.1 viene mostrato il layout in memoria di un generico programma C++.

I puntatori risiedono nello `stack` e puntano a variabili allocate nello `heap`. Hanno una dimensione fissa di 1 *word*, che equivale a 16, 32 o 64 bit in base all'architettura hardware per la quale il programma viene compilato.

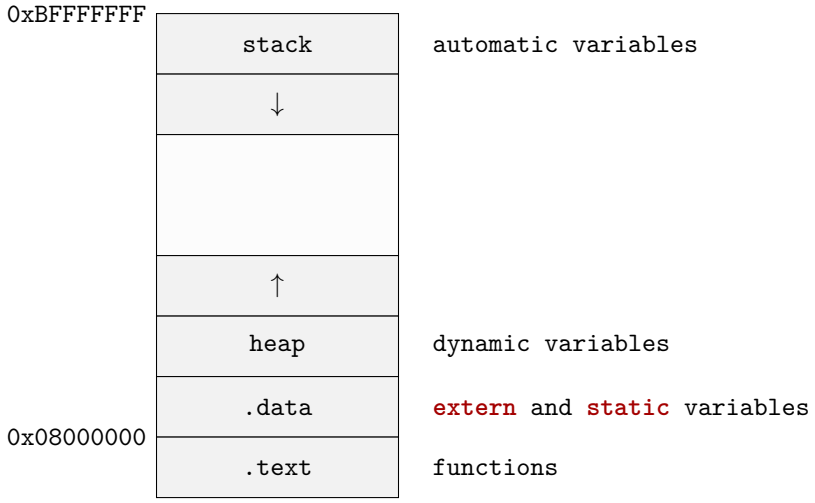


Figura 5.1: Layout in memoria di un generico programma C++.

5.3 Incapsulamento

Incapsulamento significa nascondere il funzionamento interno di una parte di un programma, in modo da proteggere le altre parti del programma dai cambiamenti che si produrrebbero in esse nel caso si decidesse di implementare la parte incapsulata in modo diverso.

Solitamente l'incapsulamento viene ottenuto sfruttando l'astrazione, fornendo un'interfaccia composta da metodi pubblici che manipolano dati nascosti. Per poter fare ciò è necessario poter specificare la *visibilità* di metodi e dati.

5.3.1 Livelli di visibilità

In C++ esistono tre livelli di visibilità, utilizzabili all'interno della dichiarazione di una `class`, `struct` o `union`:

- `public` indica che quella sezione della dichiarazione è visibile da qualunque punto del codice;
- `protected` indica che quella sezione è visibile solo all'interno della classe stessa e all'interno di classi che ereditano da essa;
- `private` indica che quella sezione è visibile solo all'interno della classe stessa.

Per `struct` e `union` la visibilità di default è `public`, mentre per `class` è `private`. All'interno della dichiarazione di una `class`, `struct` o `union`, non è necessario che le sezioni con stessa visibilità siano contigue:

```
class C {
  // private section
  ...
public:
  // This part is public
  ...
```

```
private:
  // Another private section
  ...
public:
  // Another public section
  ...
};
```

5.3.2 Dichiarazioni friend

Le dichiarazioni `friend` permettono di concedere l'accesso alle sezioni `protected` e `private` di una classe ad altre classi, metodi `public` di altre classi o funzioni:

```
class B;  // Forward declaration

class A {
  int i;
public:
  int f(int n, B& b);
  ...
};

class B {
  int i;
public:
  friend int A::f(int, B&);
  ...
};

int A::f(int n, B& b) { return i + b.i + n; }
```

Se una classe `B` ha dichiarata una classe `A` `friend`, tutto il codice scritto in `A` può accedere alle parti `private` e `protected` di `B`.

Questo meccanismo è usato quando una coppia di classi sono strettamente connesse (ad esempio matrici e vettori).

5.4 Ereditarietà

L'*ereditarietà* è la capacità di riusare la definizione di un tipo per definirne un altro. Nel codice seguente, la classe `Derived` eredita da `Base1`, ..., `BaseN` (l'ereditarietà multipla e le sue conseguenze sono spiegate nella sezione 5.4.6):

```
class Base1 { ... };
...
class BaseN { ... };
class Derived : Base1, ..., BaseN { ... };
```

Una classe derivata può diventare a sua volta la base di un'altra classe; ciò permette la creazione di gerarchie di classi.

5.4.1 Costruttori

I campi delle classi base possono (devono) essere inizializzati nel costruttore della classe derivata aggiungendo una chiamata al relativo costruttore nella *member initializer list* del costruttore della classe derivata:

```
class Base {
  int b;
public:
  Base(int v = 0) { b = v; }
};

struct Derived : Base {
private:
  int d;
public:
  Derived() : Base(), d (0) {}
  Derived(int bb, int dd) : Base(bb), d (dd) {}
};
```

La chiamata al costruttore della classe Base, se non messa esplicitamente dal programmatore, viene comunque inserita nell'initialiazer list della classe Derived. Se ad esempio:

```
struct Base {
        Base() { cout << "Base ";}
};
struct Derived :  Base {
        Derived() {cout << "Derived";}
};
```

Quando costruisco una instanza della classe Derived, avrò l'invocazione implicita del costruttore della classe Base.

```
Derived d; // output: Base Derived
```

Se la classe base ha un *default constructor*, sarà inserito quello nel costruttore della classe derivata. Questo in ogni caso crea una catena di chiamata dei costurttori.

5.4.2 Distruttori

I distuttori in C++ sono chiamati in ordine inverso rispetto a costruttori (prima quelli della classe Derivata poi quello della classe base).

Se si prevede che una certa classe possa essere utilizzata come classe base, è bene dichiarare il suo distruttore come `virtual`: in questo modo i distruttori di tutte le classi che ereditano da essa conterranno un'invocazione *implicita* del distruttore della classe base. Ciò garantisce una corretta deallocazione di tutte le risorse che un oggetto potrebbe aver acquisito.

```
struct Base {
        virtual ~Base(){cout << "Base ";}
};
struct Derived :  Base {
        ~Derived(){cout << "Derived ";}
};
```

Tabella 5.1: Visibilità dei membri della classe base in una classe derivata.

Visibilità nella classe base	Tipo di ereditarietà		
	`private`	`protected`	`public`
`private`	`private`	`private`	`private`
`protected`	`private`	`protected`	`protected`
`public`	`private`	`protected`	`public`

Grazie al fatto che il distruttore è `virtual`, viene chiamato quello corretto:

```
Base* d = new Derived; delete d;// output: Derived Base
```

Se non avessi messo virtual avrei avuto la chiamata solo del distruttore base.

5.4.3 Visibilità ed ereditarietà

Anche l'ereditarietà prevede la possibilità di specificare un livello di visibilità (vedi tabella 5.1). Da notare che i membri `private` della classe base sono presenti anche nella classe derivata, ma non c'è modo di accedervi.

Per `struct` e `union` l'ereditarietà è di default `public`, mentre è `private` per le `class`.

Una classe è sottotipo di un'altra classe solo se eredita da quest'ultima con visibilità `public`. Quindi l'ereditarietà privata è utile se si vuole nascondere parte delle funzionalità della classe base.

È comunque possibile modificare la visibilità dei membri ereditati in modo specifico per ogni membro:

```
#include <iostream>
using namespace std;

class Base {
  int x;
public:
  int y;
  void f();
  void f(int);
};

class CDerived : Base {   // Private inheritance
public:
  Base::x;  // Error! x is not visible inside CDerived
  Base::y;  // Ok, y is public
  Base::f;  // Ok, both overloaded members exposed
};

struct SDerived : Base {  // Public inheritance
private:
  Base::y;  // Ok, y is private
};
```

```
int main() {
  CDerived cd;
  cout << cd.y << endl;  // Ok
  SDerived sd;
  cout << sd.y << endl;  // Error!
  Base* b = &sd;
  cout << b->y << endl;  // Ok
}
```

5.4.4 Differenze con Java

- L'ereditarietà in Java non cambia il livello di protezione dei membri nella classe base.
- In Java non si può specificare il tipo di ereditarietà.
- I metodi ridefiniti nella classe derivata non possono ridurre l'accesso ai metodi nella classe base. Per esempio, se c'è un metodo pubblico nella classe base e lo si vuole ridefinire, il metodo ridefinito dovrà essere pubblico.

```
public class A {
  private void pri() {}
  protected void pro() {}
  public void pub() {}
}

class B extends A {
  @Override public void pri() {}   // Error! pri is not
      visible
  @Override public void pro() {}   // Ok, can increase
      visibility
  @Override private void pub() {}  // Error! Cannot reduce
                                   // visibility
}
```

5.4.5 Ridefinizione

In C++ una classe derivata può *ridefinire* un membro della propria classe base:

```
class X {
  int i;
public:
  X() { i = 0; }
  void set(int ii) { i = ii; }
  int permute() { return i = i * 47; }
};

class Y : public X {
  int i;              // Different from X::i
public:
  Y() { i = 0; }
  int change() {
```

```
    i = permute();     // Different name call
    return i;
  }
  void set(int ii) {   // Redefinition
    i = ii;
    X::set(ii);        // Same-name call
  }
};
```

Normalmente, dichiarando un membro in una classe derivata con lo stesso nome di un membro di una classe base si effettua una *ridefinizione*; se invece il membro nella classe base è una funzione dichiarata `virtual` ciò risulta in un *override*, come consueto in Java (le funzioni `virtual` saranno approfondite nelle sezioni successive). La differenza tra *ridefinizione* e *override* sta nel fatto che la prima viene risolta a *compile-time*, eventualmente utilizzando l'operatore `ClassName::`, mentre il secondo viene risolto a *run-time*, permettendo il *dynamic dispatch* e quindi il polimorfismo.

Da notare che una funzione ridefinisce quella della classe base semplicemente se ha lo stesso nome (non è necessario che la lista dei parametri sia diversa):

```
struct X {
  int f() { return 42; }
};

struct Y : X {
  int f(int i) { return i * 13; }  // Redefinition
}
```

5.4.6 Ereditarietà multipla

Una classe derivata può avere più classi base:

```
class A { ... };
class B { ... };
class C { ... };
class X : public A, private B, C { ... };
```

Questa tecnica è molto utile se si vuole riusare il codice da più classi; introduce però delle possibili ambiguità come, ad esempio, *name clash* e *member duplication*.

Name clash

Consiste nell'impossibilità, da parte del compilatore, di determinare l'esatto membro che si sta cercando di "utilizzare":

```
struct A { virtual void f() { ... } };
struct B { virtual void f() { ... } };
struct C : A, B { ... };

int main() {
  C* p = new C();
  p->f();  // Error! A::f or B::f?
}
```

Per risolvere queste ambiguità, in caso di *name clash* il programma deve specificare l'esatto membro a cui si sta cercando di accedere:

```
...
C* p = new C();
p->A::f();  // Select the f member inherited from A
```

oppure

```
...
struct C : A, B { virtual void f() { A::f(); } };

int main() {
  C* p = new C();
  p->f();  // Calls C::f, which in turn calls A::f
}
```

Member duplication

Una classe derivata può ereditare una classe base più di una volta, in modo indiretto:

```
class L { ... };
class B1 : public L { ... };
class B2 : public L { ... };
class D : public B1, public B2 { ... };
```

In questo modo un oggetto della classe `D` contiene due diverse istanze della classe `L`.

Questo problema, detto *problema del diamante*, può essere risolto dichiarando `L` come classe base `virtual` sia per `B1` che per `B2`:

```
class L { ... };
class B1 : virtual public L { ... };
class B2 : virtual public L { ... };
class D : public B1, public B2 { ... };
```

Ciò però può essere fonte di problemi nelle conversioni tra tipi.

5.5 Polimorfismo

In C++ il polimorfismo può essere di due tipi:

run-time È il polimorfismo per come viene solitamente inteso; simile al polimorfismo in Java.

compile-time Si basa sulla scrittura di codice parametrico, dove i valori dei parametri devono essere disponibili al momento della compilazione. È un meccanismo simile ai *generics* di Java, ma più potente (e più complesso).

5.5.1 Polimorfismo run-time

Funzionamento

Si può ottenere un comportamento "polimorfico" da un determinato oggetto invocando su di esso un suo metodo `virtual` tramite un puntatore o un riferimento:

```
#include <iostream>
using namespace std;

struct A {
  void m() { cout << "A::m" << endl; }
  virtual void f() { cout << "A::f" << endl; }
};

struct B : A {
  void m() { cout << "B::m" << endl; }
  virtual void f() { cout << "B::f" << endl; }
}

int main() {
  B b;
  A a = b;
  b.f();        // Prints 'B::f'
  a.f();        // Prints 'A::f'
  A& ra = b;
  ra.m();       // Prints 'A::m'
  ra.f();       // Prints 'B::f'
  ra.A::f();    // Prints 'A::f'
  A* pa = b;
  pa->m();      // Prints 'A::m'
  pa->f();      // Prints 'B::f'
  pa->A::f();   // Prints 'A::f'
}
```

Essendo che in `A` il metodo `f()` è dichiarato come `virtual`, definirlo in `B` con la stessa segnatura dà origine ad *overriding* invece che a *ridefinizione* (vero anche nel caso in cui in `B` non fosse esplicitamente dichiarato come `virtual`). Quello che accade è che il metodo da eseguire viene scelto in base al tipo *run-time* dell'oggetto puntato da `A*` (o riferito da `A&`).

Il polimorfismo può essere utilizzato quando una classe eredita pubblicamente da un'altra classe che implementa almeno un metodo `virtual`. Non è necessario che una classe ridefinisca tutti i metodi `virtual` della superclasse.

```
struct Color { ... };
struct Line { ... };

struct Point {
  Point(int xx, int yy) : x (xx), y (yy) {}
  int const x;
  int const y;
  virtual Point* move(int dx, int dy) {
    return new Point(x + dx, y + dy);
  }
  virtual Line line(Point& p) {
    double m = double(y - p.y) / (x - p.x);
    double q = y - m*x;
    return Line(m, q);
  }
};
```

```
struct ColorPoint : Point {
  ColorPoint(int xx, int yy, Color cc) : Point(xx, yy), c (cc)
      {}
  Color const c;
  // Covariant return type
  ColorPoint* move(int dx, int dy) {
    return new ColorPoint(x + dx, y + dy, c);
  }
};

struct Circle : private Point {
  // Inherited x and y are the center of the Circle
  Circle(Point p, int r) : Point(p), radius (r) {}
  int const radius;
};

int main() {
  Point p (4, 5);
  ColorPoint cp (2, 3, {});
  Point* pp = &cp;
  pp = pp->move(1, 1);  // pp points to a ColorPoint
  Line l = pp->line(p);
  Circle c (p, 5);
  pp = &c;  // Error! Point is a private base of Circle
}
```

Covarianza del tipo ritornato

Come si può notare nell'esempio precedente, se per i tipi `A` e `B` vale che `B <: A` ed esiste un metodo `A::f` dichiarato `virtual` che ritorna un puntatore o un riferimento di tipo `T`, allora `f` può essere ridefinito in `B` con tipo ritornato `U*` (o `U&`), dove `U <: T`.

Implementazione

L'utilizzo di funzioni `virtual` introduce un piccolo *overhead* al momento dell'invocazione del metodo. Inoltre, se una classe ha (o eredita) almeno un metodo `virtual` allora la sua occupazione di memoria aumenta della dimensione di un puntatore.

In fig. 5.2 infatti si può notare che il polimorfismo viene implementato aggiungendo all'oggetto un puntatore (`vptr`) ad una tabella dei metodi virtuali (`vtable`) che contiene gli indirizzi in memoria dell'implementazione dei metodi relativa alla classe di quell'oggetto.

Puntatore `this`

Il codice dei metodi di una classe viene compilato aggiungendo alla lista dei parametri il puntatore all'oggetto su cui il metodo viene invocato:

```
// Code
struct S {
  int x;
  int f(int i) { return x * i; }
```

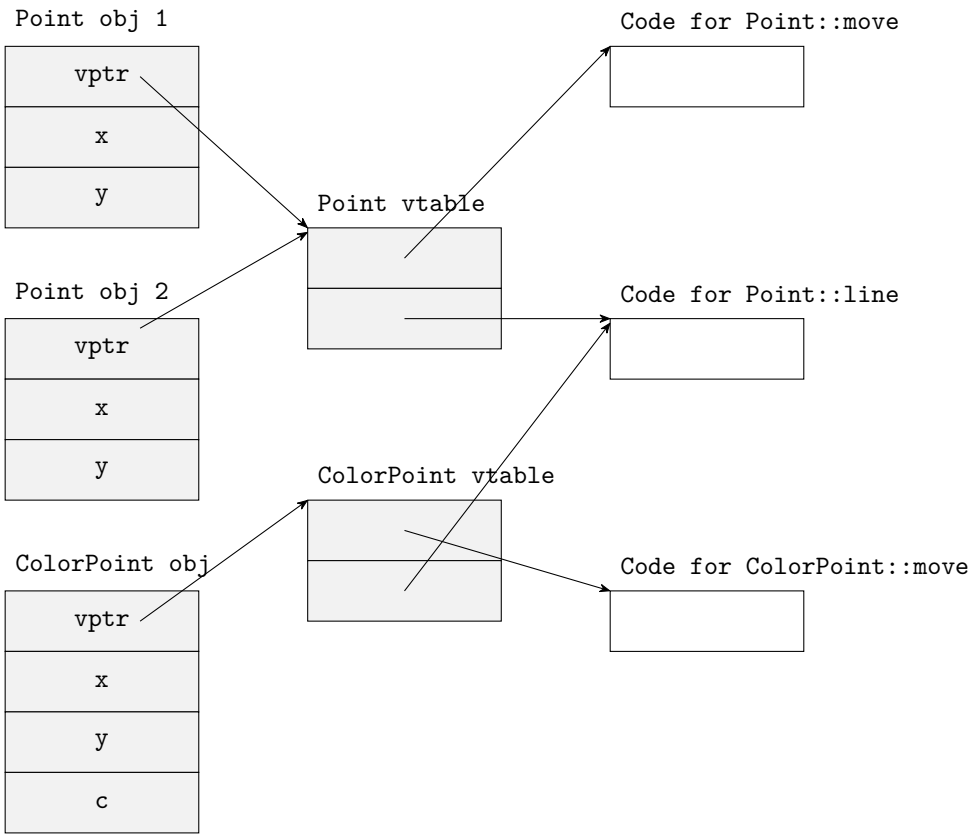


Figura 5.2: Rappresentazione del layout in memoria di oggetti con metodi `virtual`.

```
  int g(int i) { return 3 * f(i); }
};

// Compiled as
S::f(S* this, int i) { return this->x * i; }
S::g(S* this, int i) { return 3 * S::f(this, i); }
```

Tale puntatore può essere utilizzato tramite la keyword `this`:

```
struct S {
  int x;
  int f(int x) { return this->x * x; }
};
```

Distruttore

Essendo che gli oggetti che dichiarano metodi `virtual` vengono solitamente usati tramite puntatori, è bene che il loro distruttore sia anch'esso dichiarato `virtual`; in questo modo il distruttore delle classi derivate viene automaticamente invocato nel caso in cui si distrugga un oggetto tramite un puntatore di tipo della classe base.

5.5.2 Polimorfismo compile-time: i template

I `template` sono una caratteristica del C++ che permette di scrivere funzioni e classi parametriche rispetto al tipo di certe variabili o rispetto a numeri interi. Vengono utilizzati principalmente per la scrittura di algoritmi e strutture dati generiche.

Template di funzioni

Un `template` viene introdotto utilizzando la corrispondente parola chiave, seguita da un certo numero di parametri racchiusi tra parentesi spigolate. I parametri possono essere di tipo (contrassegnati indifferentemente da `typename` o `class`) o interi (`bool`, `int`, ... ed `enum`erativi).

Esempio:

```
void swap_int(int& x, int& y) {
  int tmp = x; x = y; y = tmp;
}

template<typename T>
void swap(T& x, T& y) {
  T tmp = x; x = y; y = tmp;
}
```

La funzione `swap_int` può essere utilizzata per scambiare tra loro i valori di due variabili intere, mentre la funzione `swap` può essere utilizzata per scambiare tra loro i valori di due variabili di tipo arbitrario:

```
int i, j;
...
swap(i, j);  // 'T' is 'int'
float f, g;
...
```

```
swap(f, g);  // 'T' is 'float'
...
swap(i, f);  // Error!
```

Se per l'invocazione di una funzione o metodo i parametri del `template` non possono essere dedotti automaticamente dal compilatore in qualche modo, allora vanno specificati tra parentesi spigolate dopo il nome della stessa:

```
template<typename T>
void swap(T& x, T& y) { T tmp = x; x = y; y = tmp; }

template<typename U, typename T>
U convert(T t) { return (U)t; }

int main() {
  int i, j;

  swap<int>(i, j);              // 'T' is 'int' (explicit)
  swap(i, j);                   // 'T' is 'int' (implicit)

  float x = convert(i);         // Error! 'U' unspecified
  float x = convert<float>(i);  // 'T' is 'int' (implicit)
                                // 'U' is 'float' (explicit)
  float y = convert<float, int>(i);  // Ok

  double z = convert<double, float>(i);
  // Ok, equivalent to
  double z = convert<double, float>((float)i);
}
```

A cosa possono servire i parametri interi? Ad esempio ad ordinare un array di dimensioni note in compilazione in modo più efficiente:

```
#include <cstddef>
template<typename T, std::size_t N>
void sort(T (&arr)[N]) {
  for (int i = 0; i < N-1; ++i)
    for (int j = i+1; j < N-1; ++j)
      if (arr[j] < arr[i]) swap(arr[i], arr[j]);
}

int main() {
  int arr[] = {4, 2, 9, 3, 7, 6};
  sort(arr);  // 'T' is 'int', 'N' is 6
}
```

Le operazioni che vengono eseguite su tipi generici (come `T`) limitano l'istanziabilità di un `template`, cioè: tutte le operazioni utilizzate all'interno di un `template` devono essere definite per i tipi per cui si sta cercando di instanziare il `template`. Ad esempio, la funzione `sort` appena definita può essere utilizzata solo per array di tipi per cui l'operatore "`<`" è definito. Questo comportamento è molto diverso da quello di Java (vedi sezione 4.15.2), con pro e contro. (Es.: cosa succede se invoco `sort` con argomento un array di `char*`?)

Template di classi

Le regole sono molto simili a quelle dei `template` di funzioni. Per l'istanziazione di una classe `template` però i parametri vanno sempre specificati esplicitamente.

```
template<typename T> class complex {
  T re, im;
public:
  complex(T const& r, T const& i) : re (r), im (i) {}
  T real() const { return re; }
  T imag() const { return im; }
}

int main() {
  complex<double> x (1.0, 2.0);
  complex y (3, 4);  // Error!
  complex<char*> z ("1.0", "6");
}
```

Specializzazione di template

I `template` possono essere specializzati, cioè ne può essere fornita una versione esplicita per un certo valore dei parametri:

```
template<typename T>
T sum(T a, T, b) { return a + b; }

template<>
bool sum<bool>(bool a, bool b) { return a || b; }
```

Ciò torna molto utile soprattutto nella definizione di algoritmi che siano più generici possibili:

```
#include <cstddef>  // std::size_t
#include <cstring>  // std::strcmp

template<typename T>
void swap(T* x, T* y) { T tmp = *x; *x = *y; *y = tmp; }

template<typename T>
int cmp(T const& a, T const& b) { return a - b; }

template<>
int cmp<char const*>(char const* const& a, char const* const&
   b) {
  return std::strcmp(a, b);
}

template<typename T, std::size_t N>
void sort(T (&arr)[N]) {
  for (unsigned i = 0; i < N-1; ++i)
    for (unsigned j = i+1; j < N; ++j)
      if (cmp(arr[i], arr[j]) > 0) swap(&arr[i], &arr[j]);
}
```

```
int main() {
  int a[] = {4, 3, 7, 6, 1, 8, 2, 9};
  char const* b[] = {"hello", "world", "from", "miami"};

  sort(a);
  sort(b);
}
```

5.6 Sottotipazione

La sottotipazione è una relazione sui tipi che consente a variabili di un tipo di essere utilizzate quando una variabile di un certo altro tipo sarebbe richiesta. Si indica di solito con il simbolo "`<:`".

La sottotipazione è un concetto diverso dall'ereditarietà; la prima infatti è una relazione tra interfacce, mentre la seconda è tra implementazioni. Questa cosa è evidente in C++, dove `B` è sottotipo di `A` solo se `B` eredita *pubblicamente* da `A`.

5.6.1 Slicing

Assegnando il valore di una variabile di tipo `B` ad una variabile di tipo `A`, dove `B <: A`, in assenza di utilizzo di puntatori o riferimenti ciò dà origine a *slicing*, cioè viene creato un nuovo oggetto di tipo `A` i cui campi sono inizializzati a partire dai valori dei campi dell'oggetto di tipo `B`:

```
struct A {
  int a;
  virtual char m() { return 'A'; }
};
struct B : A {
  int b;
  virtual char m() { return 'B'; }
};
int main() {
  B b; b.a = 4; b.b = 2;
  char c; int i;

  A* pa = &b;
  c = pa->m();        // c == 'B'
  i = pa->a;          // i == 4
  i = ((B*)pa)->b;    // i == 2

  A a = b;            // Slicing!
  c = a.m();          // c == 'A'
  i = a.a;            // i == 4
  i = ((B)a).b;       // Error!
}
```

In particolare, nell'ultima riga della funzione `main` il compilatore si lamenta perché non c'è nessuno modo in cui una variabile di tipo `B` possa essere inizializzata a partire da una variabile di tipo `A`.

5.6.2 Classi astratte

Una classe è astratta quando ha almeno un metodo che sia *pure* `virtual`:

```
struct S {
  virtual void pure_virtual_method() = 0;
}
```

L'indicazione "`= 0`" indica che nessuna implementazione sarà da considerare per il metodo dichiarato.

Le classi astratte sono utili per la creazione di interfacce. Una classe astratta non può essere istanziata; ciò significa che nessun oggetto di quella classe potrà essere creato, ma ci potranno essere solo puntatori o riferimenti di quel tipo. Ne consegue che una classe che eredita da una classe astratta, per poter essere istanziata dovrà implementare tutti i metodi *pure* `virtual` ereditati.

5.7 Eccezioni

La gestione delle eccezioni in C++ è simile a quella in Java, con la differenza che in C++ qualsiasi valore può essere "lanciato":

```
#include <iostream>
using namespace std;

int ratio(int a, int b) {
  if (b == 0)
    throw a;
  return a / b;
}

int main() {
  int a = 1, b = 2;
  try {
    int i = ratio(a, b);
    i = ratio(a, i);
  } catch (int x) {
    cout << "Cannot divide " << x << " by 0!" << endl;
  } catch (...) {
  }
}
```

Siccome qualsiasi cosa può essere lanciata, il blocco `catch (...)` permette di catturare qualsiasi cosa. È comunque buona norma lanciare solo oggetti che siano sottoclassi della classe standard `std::exception` dichiarata nell'header `<exception>`.

5.8 STL

La *Standard Template Library* (STL) è una libreria software molto efficiente inclusa nella libreria standard del C++ e definisce strutture dati generiche ("contenitori"), iteratori e algoritmi generici. La libreria è strutturata in modo che gli algoritmi non operino direttamente sulle strutture dati, ma su iteratori che fanno riferimento a tali strutture dati; in questo modo una sola implementazione di un certo algoritmo

Tabella 5.2: Corrispondenze principali tra Standard Template Library (STL) e Java Collection Framework (JCF).

STL	JCF
`std::vector`	`ArrayList`
`std::list`	`LinkedList`
`std::deque`	`ArrayDeque`
`std::queue`	`Queue`
`std::priority_queue`	`PriorityQueue`
`std::stack`	`Stack`
`std::map`	`TreeMap`
`std::multimap`	—
`std::set`	`TreeSet`
`std::multiset`	—
`std::unordered_map`	`HashMap`
`std::unordered_multimap`	—
`std::unordered_set`	`HashSet`
`std::unordered_multiset`	—

può essere utilizzata su diversi tipi di contenitori, a loro volta generici rispetto al tipo di dato contenuto.

5.8.1 Contenitori

Un contenitore è una collezione di valori aventi tutti lo stesso tipo. Possono essere divisi in quattro categorie:

- liste: `vector`, `list`, `deque`;
- adattatori: `queue`, `priority_queue`, `stack`;
- associativi: `map`, `multimap`, `set`, `multiset`;
- associativi senza ordine: `unordered_map`, `unordered_multimap`, `unordered_set`, `unordered_multiset`.

Contenitori non associativi

Il contenitore più noto e utile è probabilmente `vector`, assimilabile ad un vettore allocato dinamicamente ad accesso random:

```
#include <vector>

struct Integer {
  int value;
  Integer(int x) : value (x) {}
};

int main() {
```

```
  std::vector<int> v (10);        // 'v' is a vector of 10 int's
  std::vector<Integer> w (10);    // Error! Integer is not
      default
                                  // constructible

  std::vector<Integer> x (5, Integer(4));
  // Ok, vector of 5 Integer initialized with 'Integer(4)'
  ...
  v[2] = 18;
}
```

Il contenuto di un `vector` può essere modificato a piacere; due operazioni che hanno costo $O(1)$ (ammortizzato) sono l'inserimento e la rimozione di elementi in coda, con le rispettive segnature:

```
template<typename T>
void std::vector<T>::push_back(T const& value);

template<typename T>
void std::vector<T>::pop_back();
```

Definisce inoltre molti altri metodi utili, come `size()-> size_type` che permette di conoscere il numero di elementi contenuti e `empty()-> bool` che permette di sapere se un `vector` sia vuoto oppure no. Tutti i contenitori della STL definiscono anche dei membri di tipo, che permettono poi agli algoritmi di funzionare indipendentemente dal contenitore:

```
namespace std {
  template<typename T> class vector {
    typedef T value_type;
    typedef value_type& reference;
    typedef unsigned int size_type;
    typedef ... iterator;
    ...
  };
}
```

I metodi `resize(size_type)-> void` e `clear()-> void` permettono di ridimensionare il contenitore.

L'accesso al singolo elemento è possibile utilizzando l'operatore `[] -> reference` (come con gli array) oppure la funzione `at() -> reference` (quest'ultima effettua *bound checking*). Essendo che queste funzioni ritornano un riferimento all'elemento, è possibile utilizzare quest'ultimo come una normale variabile di quel tipo:

```
std::vector<int> v (10);
std::vector<std:string> s (2);
...
v.at(4) = v.at(3) + 1;
...
s[0] += s[1];  // s[0] is now 's[1]' concatenated to 's[0]'
```

Esistono anche altri operatori che funzionano bene anche su `vector`:

```
std::vector<int> v (10), w (20);
...
```

```
v = w;          // 'v' old content is gone.
                // Now 'v' is a copy of the elements in 'w'
...
if (v == w)     // Same size? Same elements?
  ...
else
  ...
```

Prestare attenzione a quando si passa un `vector` ad una funzione: passandolo per valore l'intero contenuto viene copiato; per questo è bene passarlo per riferimento (eventualmente `const` se la funzione che lo riceve deve non poterlo modificare).

È possibile anche rimuovere o inserire elementi in posizioni arbitrarie, utilizzando rispettivamente le funzioni `erase(iterator)-> iterator` e `insert(iterator, T const& value)-> iterator`:

```
std::vector<int> v (10);
...
v.erase(v.begin()+4);       // Removes the 4th element
...
v.insert(v.begin()+6, 8);   // Inserts '8' as 6th element
```

Tuttavia su di un `vector` queste operazioni sono poco efficienti, perché richiedono di *shiftare* tutti gli elementi a valle della posizione rimossa o inserita, quindi si consiglia di usare un'altra struttura dati nel caso in cui questo tipo d'operazione debba essere usata di frequente.

La classe `list` ha un'interfaccia molto simile a quella di `vector`, con qualche differenza, dovuta al fatto che è implementata come lista di nodi bidirezionale:

- `list` non permette l'accesso ad elementi in posizione random, ma solo un accesso sequenziale (tramite *iteratori*);
- `list` permette l'inserimento e la rimozione di elementi in posizione arbitraria in tempo $O(1)$, quando per `vector` il tempo è $O(n)$;
- `list` definisce qualche operazione aggiuntiva che è molto efficiente per come è impelemntata, come il *merge* di due `list` o la rimozione di elementi in base ad un predicato.

Contenitori associativi

I contenitori di tipo associativo sono diversi da quelli non associativi, in quanto sono pensati per rendere più efficiente la ricerca di un valore in base ad una determinata chiave:

- le versioni "base" (`map`, `multimap`, `set`, `multiset`) hanno prestazioni logaritmiche, ma mantengono un ordinamento tra gli elementi contenuti;
- le versioni `unordered` hanno prestazioni $O(1)$ (ammortizzate) ma non mantengono i dati ordinati.

Forniscono operazioni per l'inserimento, la rimozione e la ricerca di valori in base ad una chiave, per cui deve essere definito:

- versione “base”: un criterio di ordinamento;
- versione `unordered`: una funzione di *hash*.

Esempio:

```
std::map<std::string, double> m;
m.insert(std::make_pair("mean", 5.3));
// Inserts (or assigns) the value '1.2' for the key '"stddev"'
m["stddev"] = 1.2;
...
// Erases the value for '"something"' (if present)
m.erase("something");
...
for ( std::map<std::string>::iterator it = m.begin();
      it != m.end(); ++it )
  std::cout << it->first << ": " << it->second << std::endl;
```

Contenitori e gerarchie di classi

Per ottenere un comportamento polimorfico da oggetti contenuti nei contenitori della STL è necessario che il contenitore non contenga gli oggetti direttamente (che darebbe luogo a *slicing*) ma i puntatori (o i riferimenti) a tali oggetti:

```
#include <vector>

class Shape {
  virtual void draw() = 0;
  ...
};

class Rectangle {
  virtual void draw() { ... }
  ...
};

int main() {
  std::vector<Shape*> v (10);
  v[0] = new Rectangle(4, 3);
  ...
  delete v[0];
}
```

5.8.2 Iteratori

Gli *iteratori* generalizzano il concetto di puntatore ad una posizione di un contenitore. Per essi sono in genere definite tutte le operazioni che ha senso eseguire con un puntatore:

- avanzamento e arretramento;
- dereferenziazione dell’elemento alla posizione puntata.

Per ogni contenitore sono in genere definite due funzioni (con varianti `const`) che restituiscono i corrispettivi iteratori:

- `begin() -> iterator`: restituisce un iteratore al primo elemento del contenitore;
- `end() -> iterator`: restituisce un iteratore all'elemento *dopo* l'ultimo;

da utilizzare nel seguente modo:

```
template < typename Container
         , typename Value = typename Container::value_type >
bool find(Container const& c, Value const& v) {
  for ( typename Container::const_iterator it = c.begin();
        it != c.end(); ++it ) {
    if (*it == v)
      return true;
  }
  return false;
}
```

L'algoritmo appena definito cerca un elemento all'interno di un contenitore, ed essendo che sfrutta il concetto di *iteratore*, è indipendente dal contenitore stesso, e può quindi essere utilizzato su ogni contenitore della STL che permetta un accesso sequenziale:

```
std::vector<int> v;
std::list<std::string> l;
...
bool i = find(v, 8);
bool s = find(l, "value");
```

Esistono diversi tipi di iteratori, in base all'efficienza e/o alla possibilità di eseguire certe operazioni sul contenitore per cui sono definiti:

- `random_access_iterator`: può avanzare e arretrare di un numero arbitrario di posizioni (in un'operazione) sola;
- `bidirectional_iterator`: può avanzare e arretrare di una posizione alla volta;
- `forward_iterator`: può avanzare di una posizione alla volta;
- `input_iterator`: può avanzare di una posizione alla volta, permette solo di "leggere" il dato puntato;
- `output_iterator`: può avanzare di una posizione alla volta, permette solo di "scrivere" il dato puntato.

Per tutti gli `iterator` esiste poi una versione `const_iterator` che punta a riferimenti `const`, e per alcuni esiste anche la versione `reverse_iterator` che permette di invertire le operazioni di avanzamento e arretramento. Nell'implementare un algoritmo generico che sfrutta gli iteratori, meno operazioni degli iteratori si sfruttano e più l'algoritmo sarà generalizzabile.

5.8.3 Algoritmi

La STL definisce molti algoritmi per la modifica dei contenitori o per la verifica di certe proprietà.[III]

Due algoritmi molto "di moda", in quanto permettono di estrarre una certa informazione da una serie di dati:

- `std::transform` (equivalente all'algoritmo *map* di altri linguaggi), permette di estrarre un certo "aspetto" da un singolo dato (es.: un membro di una classe);
- `std::accumulate` (equivalente all'algoritmo *reduce* di altri linguaggi), permette di "riassumere" un'informazione su una serie di dati in un unico valore.

Permettono quindi di implementare lo schema di programmazione *MapReduce* utile quando si devono estrarre dati da grandi collezioni.

Esempio:

```
#include <algorithm>  // std::transform
#include <numeric>    // std::accumulate
#include <vector>     // std::vector

struct Student {
  ...
  int age;
};

int getAge(Student const& s) { return s.age; }

int main() {
  std::vector<Student> v;
  ...
  std::vector<int> ages (v.size());
  std::transform(v.begin(), v.end(), ages.begin(), getAge);
  double sum = std::accumulate(ages.begin(), ages.end(), 0.0);
  double mean_age = sum / ages.size();
  ...
}
```

5.9 Alcune aggiunte non relative agli oggetti

5.9.1 Tipo bool

Anche in C si possono usare i tipi boolean importando la libreria `stdbool`:

```
#include <stdbool.h>
...
bool a = true;
...
```

Tuttavia in C queste variabili bool sono variabili intere (0 per falso e non zero per vero) introdotte con un typedef. Possono essere usate in modo non corretto. Il C++

[III] Per una lista completa si rimanda a questo link.

invece introduce il tipo primitivo `bool` con le sue costanti true e false. Per ragioni di compatibilità però può essere usato con gli interi. Per esempio:

```
bool b1, b2; // due variabili booleane
bool b3 = true;

int x1 = 10, x2 = 20, m = 2;
b1 = x1 == x2; // assegnamento a confronto: false
b2 = x1 < x2; // true

cout << "b1 is = " << b1 << "\n"; // stampa 0
cout << "b2 is = " << b2 << "\n"; // stampa 1

if (b3)
        cout << "Yes" << "\n";
else
        cout << "No" << "\n";

int x3 = false + 5 * m - b3;
cout << x3;
```

5.9.2 Tipi riferimenti

vedi slides

5.9.3 Copy constructor e assignment operator

vedi slides

5.9.4 Tipo string

vedi slides

6 Scala

6.1 Introduzione

Scala è un linguaggio object-oriented, tipizzato staticamente, che si esegue sulla Java Virtual Machine. La peculiarità di Scala è che utilizza una programmazione funzionale con caratteristiche dinamiche.

Scala può essere usato al posto di Java, usa librerie di Java, usa tools di Java e ha degli IDE di supporto accettabili.

Cosa c'è di sbagliato in Java? E' un po' troppo prolisso:

```
Thing thing = new Thing();
```

Cosa c'è di buono in Java?

- molto popolare;
- object-oriented;
- tipizzazione forte;
- numerose librerie;
- piattaforma indipendente molto ottimizzata (JVM).

6.1.1 Scala è come Java, ad eccezione di quando non lo è (Y)

- Java è un buon linguaggio e Scala lo è anche di più
- Per ogni differenza tra i due linguaggi c'è una ragione, ovvero non ci sono cambiamenti fatti solo per rendere i due linguaggi diversi.
- Scala e Java sono quasi completamente interoperabili:
  - se chiami Java da Scala non ci sono problemi;
  - se chiami Scala da Java ci sono alcune restrizioni, ma è per lo più ok;
  - Scala compila i file `.class` (quasi tutti) e può essere eseguito sia con comandi Scala che con comandi Java.
- Capire Scala aiuta a comprendere le ragioni dei cambiamenti nella programmazione, e che cosa Scala sta cercando di realizzare

Consistenza

In Java, tutti valori sono oggetti, esclusi i tipi primitivi. I numeri e i booleani sono primitivi per ragioni di efficienza, quindi dobbiamo trattarli in modo diverso.

In Scala, tutti i valori sono oggetti. Il compilatore li converte in primitivi quindi l'efficienza non è persa.

Java ha operatori (`+`, `-`, `<`, ...) e metodi con differente sintassi.

In scala, gli operatori sono solo metodi e in molti casi si possono utilizzare entrambe le sintassi.

6.1.2 Scala è conciso

```
// Type inference
val sum = 1 + 2 + 3
val nums = List(1,2,3)
val map = Map("abc" -> List(1,2,3))
// Explicit types
val sum: Int = 1 + 2 + 3
val nums: List[Int] = List(1,2,3)
val map: Map[String, List[Int]] = ...
```

La programmazione con Scala è più ad alto livello, per esempio per controllare se una stringa ha almeno un carattere maiuscolo basta l'istruzione:

```
val hasUpperCase = name.exists(_.isUpperCase)
```

mentre Java ci obbligherebbe ad una programmazione più lunga (ciclo `for`, ...).

Stesso discorso se pensiamo di progettare una classe `Person` con gli attributi `name` ed `age` e i metodi *getter* e *setter*.

```
// Scala
class Person(var name: String, private var _age: Int) {
  def age = _age              // getter for age
  def age_=(newAge: Int) {    // setter for age
    println("Changing age to: " + newAge)
    _age = newAge
  }
}
```

```
// Java
public class Person {
  private String mName;
  private int mAge;
  public Person(String name, int age) {
    mName = name;
    mAge = age;
  }
  public String getName() {
    return mName;
  }
  public int getAge() {
    return mAge;
  }
  public void setName(String name) {
```

```
    mName = name;
  }
  public void setAge(int age) {
    mAge = age;
  }
}
```

Differenza tra variabili (`var`) e valori (`val`)

```
var foo = "foo"
val bar = "bar"
foo = "bar"  // Allowed
bar = "foo"  // Not allowed
```

In scala esiste l'identificatore `null`? Sì.

> Lo chiamo il mio errore da un miliardo di dollari. Fu l'invenzione del riferimento *null* nel 1965. A quel tempo stavo progettando il primo type system completo per i riferimenti in un linguaggio object-oriented (ALGOL W). Il mio scopo era assicurarmi che tutti gli usi dei riferimenti fossero assolutamente sicuri, con un controllo performante fatto automaticamente dal compilatore. Ma non potei resistere alla tentazione di mettere un riferimento nullo, semplicemente perché era molto facile da implementare. Questo ha portato innumerevoli errori, vulnerabilità e crash di sistemi che hanno probabilmente causato un miliardo di dollari di pena e danni negli ultimi 40 anni.
> — Tony Hoare

In Java alcuni metodi suppongono che si ritorni un oggetto che potrebbe essere `null`. Si può controllare che sia `null`, si può mettere il metodo in un costrutto `try...catch` oppure accertarsi che il metodo non possa ritornare tale valore.

Scala ha il tipo `null` solo per poter comunicare con Java. In Scala, se un metodo può non ritornare nulla si scrive che ritorna un `Option[Object]`, che può essere o `Some[Object]` (l'oggetto) oppure `None`. Questo forza l'utilizzo di un *match* statement, ma solo dove è realmente necessario.

6.1.3 Object-oriented, funzionale

Cosa è la programmazione multi-paradigma?

Essa fornisce "un contesto in cui i programmatori possano lavorare in una varietà di stili, con costrutti di diversi paradigmi liberamente mescolati." — Tim Budd.

Imperativo vs. dichiarativo

Perché imparare la programmazione multi-paradigma?

> I risultati della ricerca della psicologia della programmazione indicano che le competenze in programmazione sono molto più fortemente legate al numero di diversi stili di programmazione compresi da una persona

> rispetto a quello che è il numero di anni di esperienza nella programmazione.
> — Tim Budd

Lo scopo della programmazione multi-paradigma è fornire un numero diverso di stili di risoluzione ai problemi in modo che il programmatore possa selezionare una tecnica di soluzione migliore in base alle caratteristiche del problema.

Perché imparare la programmazione multi-paradigma? La programmazione imperativa e object-oriented hanno caratteristiche della programmazione funzionale (sempre più), per esempio le higher order functions. Inoltre nuovi linguaggi multi-paradigma sono comparsi e quindi è bene conoscerli.

Linguaggi funzionali

I linguaggi funzionali più noti sono ML, OCaml e Haskell.

Sono considerati:

- usati solo in ambito accademico (per lo più vero);
- difficili da imparare (vero);
- la soluzione a tutti i problemi di programmazione concorrente (esagerato ma non completamente falso).

Scala è un linguaggi funzionale impuro, nel senso che puoi programmare in modo funzionale ma non è obbligatorio.

La speranza è che Scala permetta alle persone di introdursi nella programmazione funzionale (FP) e gradualmente di usarla. Così come il C++ introduce l'utilizzo della programmazione object-oriented. Inoltre un po' di programmazione funzionale rende le cose un po' più semplici.

La FP è un modo diverso di pensare riguardo la programmazione e non è facile da padroneggiare. Ma la maggior parte delle persone che riesce a padroneggiarlo non torna più indietro.

6.1.4 Dinamicità

Scala è dinamico! (In realtà non lo è, ma ha tante caratteristiche tipiche dei linguaggi dinamici.)

```
def doTalk(any: {def talk: String}) {
  println(any.talk)
}

class Duck { def talk = "Quack!" }
class Dog  { def talk = "Bark!"  }

doTalk(new Duck)
doTalk(new Dog)
```

Nella *tipizzazione dinamica* la semantica di un oggetto è determinata dall'insieme corrente dei suoi metodi e delle sue proprietà anziché dal fatto di estendere una particolare classe o implementare una specifica interfaccia. Permette quindi il *polimorfismo* senza ereditarietà.

6.1.5 Altre caratteristiche

Valori di default dei parametri

```
def hello(foo: Int = 0, bar: Int = 0) {
  println("foo: " + foo + ", bar: " + bar)
}

hello()       // foo: 0, bar: 0
hello(1)      // foo: 1, bar: 0
hello(1, 2)   // foo: 1, bar: 2
```

Parametri con nome

```
def hello(foo: Int = 0, bar: Int = 0) {
  println("foo: " + foo + ", bar: " + bar)
}

hello(bar=6)          // foo: 0, bar: 6
hello(foo=7)          // foo: 7, bar: 0
hello(foo=8, bar=9)   // foo: 8, bar: 9
```

Tutto ritorna un valore

```
val a = if (true) "yes" else "no"
val b = try { "foo" } catch { case _ => "error" }
val c = { println("hello"); "foo" }
```

Lazy vals

```
lazy val foo = {
  println("init")
  "bar"
}

val a = foo   // Prints "init"
val b = foo   // Prints nothing
```

Funzioni annidate

```
def outer() {
  var msg = "foo"
  def one() {
    def two() {
      def three() { println(msg) }
      three()
    }
    two()
  }
  one()
}
```

Gli argomenti non sono valutati al punto della chiamata della funzione, anzi sono valutati ad ogni uso dentro la funzione:

```
def f(x: Int, y: Int) = x          // Pass by value
def g(x: => Int, y: => Int) = x    // Pass by name
```

Quale dei due metodi è più veloce?

```
def test(x: Int, y: Int) = x * x
def test(x: => Int, y: => Int) = x * x

test(2, 3)
// Call by value : 2*2 = 4
// Call by name  : 2*2 = 4

test(3+4, 8)
// Call by value : test(7, 8)  = 7*7      = 49
// Call by name  : (3+4)*(3+4) = 7*(3+4) = 7*7 = 49

test(7, 2*4)
// Call by value : test(7, 14) = 7*7 = 49
// Call by name  : 7*7         = 49

test(3+4, 2*4)
// Call by value : test(7, 2*4) = test(7,8) = 7*7 = 49
// Call by name  : (3+4)*(3+4)  = 7*(3+4)   = 7*7 = 49
```

defs, vals e vars

`def` Definisce le funzioni con parametri, l'espressione è valutata ad ogni chiamata

`val` Definisce valori costanti, l'espressione viene valutata immediatamente all'inizializzazione

`var` Definisce la locazione di memoria in cui i valori possono essere cambiati con degli assegnamenti. L'espressione è valutata immediatamente all'inizializzazione.

```
var msg = "Hello"
msg += " world"          // Ok, msg = "Hello world"
msg = 5;                 // Errore, msg é di tipo String
```

```
val msg = "Hello world" // msg é immutabile
msg += "!"              // Errore
```

Perchè si usa l'immutabilità?

- Gli oggetti immutabili sono automaticamente *thread-safe*;
- il compilatore può decidere meglio sui valori immutabili, quindi migliora l'ottimizzazione.

> Inizia con l'immutabilità, poi usa la mutabilità dove la trovi appropriata.
> — Steve Jenson

6.2 Object-orientation in Scala

Scala object-oriented è:

- basata sulle classi;
- singola ereditarietà;
- si possono definire oggetti *singleton* facilmente;
- *traits*, tipi composti e viste permettono una maggiore flessibilità.

```
// I parametri del costruttore diventano membri pubblici
class Person(val name: String, var age: Int) {
  if (age < 0) { ... }
  ...
}

var p = new Person("Peter", 21)
p.age += 1
```

In scala il costruttore primario è il corpo della classe e i suoi parametri sono presentati dopo il nome della classe. Creiamo poi variabili (campi) usando sia `val` che `var`. Usando `val` creiamo variabili immutabili che possiamo quindi solo leggere.

```
class Person(n: String) {
  val name = n
  def getName() = name
}
class Person(val name: String) {
  def getName() = name
}
class Person(var name: String) {
  def getName() = name
}
```

6.2.1 Object

creiamo il singleton `object Person`, questo sarà un oggetto "compagno" della classe `Person`:

```
object Person {
  def defaultName() = "nobody"
}
class Person(val name: String, var age: Int) {
  def getName(): String = name
}

val singleton: Person = Person  // Person é un object
```

6.2.2 Estensione delle classi

```
class Point(xc: Int, yc: Int) {
  val x: Int = xc
  val y: Int = yc
  def move(dx: Int, dy: Int): Point = new Point(x+dx, y+dy)
}
class ColorPoint(u: Int, v: Int, c: String) extends Point(u, v
   ) {
  val color: String = c
  def compareWith(p: ColorPoint): Boolean =
    (p.x == x) && (p.y == y) && (p.color == color)
  override def move(dx: Int, dy: Int): ColorPoint =
    new ColorPoint(x+dx, y+dy, color)
}
```

`ColorPoint` aggiunge un nuovo metodo e fa l'`override` di un altro metodo con più o meno le stesse regole di Java.

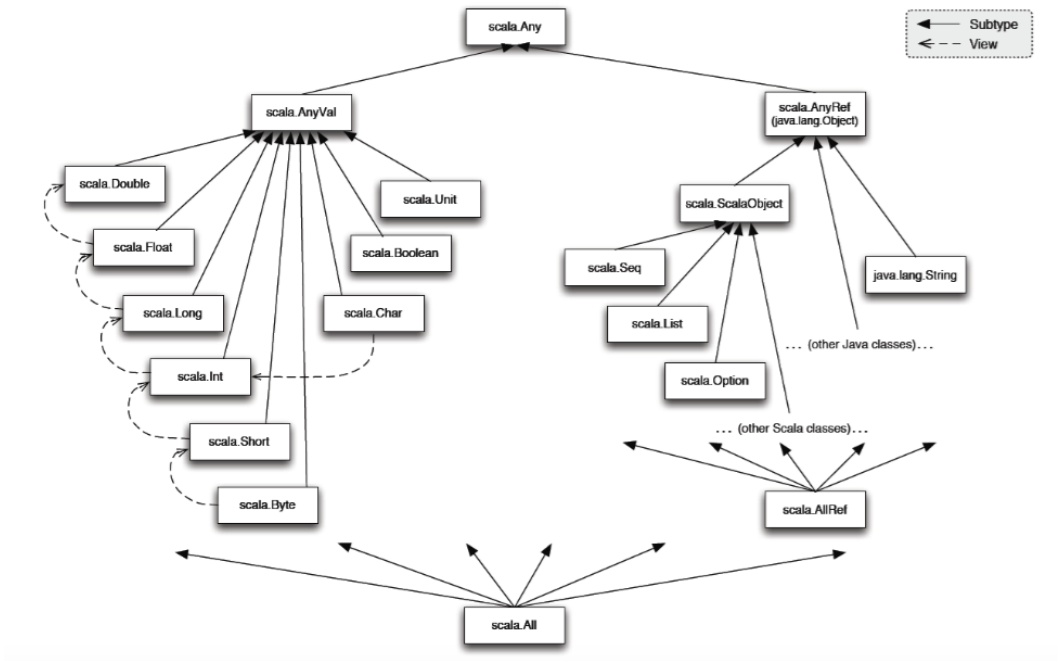
6.2.3 Classi astratte (esempio)

```
abstract class Greeter {
  val message: String
  def sayHi() = println(message)
}
class BerghemGreeter extends Greeter {
  val message = "Alura"
}

object Prova {
  val greeter = new BerghemGreeter()
  greeter.sayHi()
}
```

6.2.4 Traits

Simili alle interfacce di Java, possono implementare dei metodi ma non possono contenere stati. Possono far derivare eredità multiple. Come risolve Scala il *problema del diamante*? In modo molto semplice: considera l'ordine in cui i `trait`s sono ereditati. Se ci sono implementazioni multiple di un membro ricevuto, l'implementazione nel supertipo che è più a destra "vince".

```
trait Similarity {
  def isSimilar(x: Any): Boolean
}

class Student extends Similarity {
  def isSimilar(x: Student) = true
}
```



6.3 Function Programming in Scala

In questa parte spieghiamo come programmare in modo *funzionale* in Scala

Function Definition

```
def add(a:Int, b:Int):Int = a + b

val m:Int = add(1, 2)

println(m)
```

Note the use of the type declaration "Int". Scala is a "statically typed" language. We define "add" to be a function which accepts two parameters of type Int and returns a value of type Int. Similarly, "m" is defined as a variable of type Int.

Function Definition

```
def fun(a: Int):Int = {
        a + 1
        a - 2
        a * 3
}

val p:Int = fun(10)
println(p)
```

Note!
There is no explicit "return" statement! The value of the last expression in the body is automatically returned.

Type Inference

```
def add(a:Int, b:Int) = a + b

val m = add(1, 2)

println(m)
```

We have NOT specified the return type of the function or the type of the variable "m". Scala "infers" that!

Non possiamo invece non dichiarare i tipi dei paramtri:

Type Inference

```
def add(a, b) = a + b

val m = add(1, 2)

println(m)
```

This does not work! Scala does NOT infer type of function parameters, unlike languages like Haskell/ML. Scala is said to do local, "flow-based" type inference while Haskell/ML do Hindley-Milner type inference

References

1. http://en.wikipedia.org/wiki/Hindley-Milner
2. http://www.scala-lang.org/node/4654

Expression Oriented Programming

```
val i = 3
val p = if (i > 0) -1 else -2
val q = if (true) "hello" else "world"

println(p)
println(q)
```

Unlike languages like C/Java, almost everything in Scala is an "expression", ie, something which returns a value! Rather than programming with "statements", we program with "expressions"

Expression Oriented Programming

```
def errorMsg(errorCode: Int) = errorCode match {
case 1 => "File not found"
case 2 => "Permission denied"
case 3 => "Invalid operation"
}

println(errorMsg(2))
```

Case automatically "returns" the value of the expression corresponding to the matching pattern.

Evaluation

Applications of parametrized functions are evaluated in a similar way as operators. Expressions are evaluated before passing their value to functions (values are passed to functions)

```
def square(x:Double) = x * x

square(2)

square(2+2)

square(square(2))
```

Evaluation of function application

Given a function application $f(e_1, \ldots, e_n)$

1. Evaluate all function arguments $(e_1, \ldots, e_n)$ from left to right. Let $v_1, \ldots, v_n$ the corresponding values.
2. Replace the function application by the function's right hand side (function body), and, at the same time
3. Replace (substitute) the formal paramters of the function by the actual arguments $v_1, \ldots, v_n$

```
def sumOfSquare(x:Double, y: Double) = square(x) +
    square(y)
```

```
sumOfSquare(3,2+2)
sumOfSquare(3,4)
square(3) + square(4)
3 * 3 + square(4)
9 + square(4)
....
```

Evaluation of function application

This scheme of expression evaluation is called the *substitution model*

The basic idea is to reduce an expression to a value

It can proved that this model can represent any algorithm (except side effect).

Termination

Does every expression reduce to a value (in a finite number of steps)? NO

```
def loop: Int = loop

loop
```

Alternative evaluation

The interpreter reduces function arguments to values before rewriting the function application.

One could alternatively apply the function to unreduced arguments.

For instance:

```
sumOfSquare(3,2+2)
square(3) + square(2+2)
3*3 + (2+2) * (2+2)
....
```

Pass by-name

The first strategy is call-by-value, the second is *call-by-name* (they are equivalent if they terminate - but CBN may terminate)

- call-by-value: evaluates every function argument only once
- call-by-name: a function is evaluated only if necessary

```
def test(x:Int, y:Int) = x * x
```

which call is fastest (fewest num. operations)?

test(2,3) same number of steps test(3+4,8) CBV faster test(7,2*4) CBN faster test(3+4,2*4) same number of steps

Call by name functions

Scala normally uses call-by-value

But if the type of a function parameter with => it uses call-by-name

Example

```
def constOne(x:Int, y: => Int) = 1
```

Using pen and paper, trace the evaluation of the following function calls for the function constOne:

constOne(1+2,loop) constOne(loop,1+2)

Recursion

- Recursion means a function can call itself repeatedly.
- Recursion plays a big role in pure functional programming and Scala supports recursion functions very well.

Consider the Eulid's algorithm

```
def gcd(a: Int, b: Int) = if (b == 0) a else gcd(b,a%b
    )
```

evaluate gcd(14,21) ...

Consider the factorial algorithm

```
def factorial(n: Int) = if (n == 0) 1 else n*factorial
    (n-1)
```

evaluate factorial(4) ...

Recursion

```
// sum n + (n-1) + (n-2) + ... + 0
def sum(n: Int): Int =
        if (n == 0) 0 else n + sum(n - 1)

val m = sum(10)
println(m)
```

Try calling the function "sum" with a large number (say 10000) as parameter! You get a stack overflow!

Tail Calls and TCO

If a function calls itself as its last action is called tail recursion. The function's stack frame can be reused (Tail Call Optimization).

Rewrite the function as tail recursion.

```
def factorial(n: Int): Int = {
        def loop(acc: Int, n: Int)=
                if ( n == 0) acc
                else loop(n*acc, n-1)
        loop(1,n)
}
```

Tail Calls and TCO

```
def sum(n: Int, acc: Int):Int =
        if(n == 0) acc else sum(n - 1, acc + n)

val r = sum(10000, 0)

println(r)
```

This is a "tail-recursive" version of the previous function - the Scala compiler converts the tail call to a loop, thereby avoiding stack overflow!

Proviamo con la segnatura semplificata e un loop interno.

Tail Calls and TCO

```
(sum 4)
(4 + sum 3)
(4 + (3 + sum 2))
(4 + (3 + (2 + sum 1)))
(4 + (3 + (2 + (1 + sum 0))))
(4 + (3 + (2 + (1 + 0))))
(4 + (3 + (2 + 0)))
(4 + (3 + 2))
(4 + 5)
(9)
------------------------------
(sum 4 0)
(sum 3 4)
```

```
(sum 2 7)
(sum 1 8)
(sum 0 9)
(9)
-------------------------------
```

Higher-Order Functions (HOF)

- Functional languages treat functions as *first-class values*
- This means that, like any other value, a function can be passed as a parameter and returned as a result
- Functions that take values and variable are called *first order functions*
- Functions that take other functions as parameters or return functions are called *higher order functions*

Summation once again!

take the sum of the integers from a and b:

```
def sumInts(a: Int, b: Int) =
if (a > b) 0 else a + sumInts(a+1,b)
```

If you want to sum the squares or cubes from a and b:

```
def sqr(x: Int) = x * x
def sumSquares(a: Int, b: Int): Int =
if (a > b) 0 else sqr(a) + sumSquares(a + 1, b)

def cube(x: Int) = x * x * x
def sumCubes(a: Int, b: Int): Int =
if (a > b) 0 else cube(a) + sumCubes(a + 1, b)
```

Summation

Exercise
Define the sum of factorial from a and b

Idea
Define a sum generic with the respect to the operation applied to each number?

```
def operation(x:Int) = ...

def sumOperation(a: Int, b: Int) =
if (a > b) 0 else operation(a) +  sumOperation(a+1,b)
```

Higher order functions

```
def sum(f: Int=>Int, a: Int, b: Int): Int =
if (a > b) 0 else f(a) + sum(f, a + 1, b)
```

"sum" is now a "higher order" function! It's first parameter is a function which maps an Int to an Int

The type "Int" represents a simple Integer value. The type `Int => Int` represents a function which accepts an Int and returns an Int.

```
def identity(x: Int) = x
def sqr(x: Int) = x * x
def cube(x: Int) = x * x * x
def fact(x:Int) = ...
println(sum(identity, 1, 10))
println(sum(sqr, 1, 10))
println(sum(cube, 1, 10))
```

Anonymous functions

Passing functions as parameters leads to the creation of many functions. Sometime is tedious. It can be avoided.

Like:

```
def name = "Angelo"; println(name)
```

can be written as

```
println("Angelo")
```

we want to define functions without an explicit name:

anonymous functions

Anonymous functions

Can be written as:

(*Parameters*) => *Body*

We can create "anonymous" functions on-the-fly! `x => x*x` is a function which takes an "x" and returns x*x

```
(x:Int)=> x *x
```

The parameter type can be omitted if the compiler can infer it:

```
x => x *x
```

```
println(sum(x=>x, 1, 10))
println(sum(x=>x*x, 1, 10))
println(sum(x=>x*x*x, 1, 10))
```

Higher order functions and recursive calls

Rewrite sum with the tail recursion?

```
def sum(f: Int => Int, a: Int, b: Int): Int = {
        def loop(a: Int, acc: Int): Int = {
                if (a > b) acc
                else loop(a + 1, acc + f(a))
        }
        loop(a, 0)
}
```

Currying

Here is the definition from Wikipedia:

In mathematics and computer science, currying is the technique of transforming a function that takes multiple arguments (or a tuple of arguments) in such a way that it can be called as a chain of functions, each with a single argument. It was originated by Moses Schonfinkel and later re-discovered by Haskell Curry.

Let's try to do this in Scala!

Currying - returning functions

```
def sumInts(a:Int, b: Int) =  sum(x=>x, 1, 10)
def sumCubes(a:Int, b: Int) =  sum(x=>x*x, 1, 10)
def sumFactorial(a:Int, b: Int) =  sum(fact, 1, 10)
```

a and b are passed to sum unchanged. Can we rewrite sum and avoid the use of a and b?

`sum` can *return* a function that takes two Ints and return an Int

```
def sum(f: Int => Int): (Int,Int) => Int = { ...}
```

sum takes a function `f` and return a function `(Int,Int) => Int`

Currying - returning functions

```
def sum(f: Int => Int): (Int,Int) => Int = {
        def sumF(a: Int, b: Int): Int = {
                if (a > b) 0
                else f(a) + sumF(a + 1, b)
        }
        sumF
}
```

Currying - stepwise application

The basic sum functions can be defined without parameters:

```
def sumInts =  sum(x=>x)
def sumCubes =  sum(x=>x*x)
def sumFactorial =  sum(fact)
```

sumInts(3,4) ...
or we could write

```
sum(x=>x)(3,4)
```

Multiple parameter list

```
def sum(f: Int => Int, a: Int, b: Int): Int =
```

Can be rewritten as:

```
def sum(f: Int => Int) : (Int, Int) => Int =
```

or equivalently, by using multiple parameter lists:

```
def sum(f: Int => Int)(a: Int, b: Int): Int =
```

the advantage wrt the first is that se can pass only one argument like sum(cube) the advantage wrt the second is that we can use a and b directly in the body

Currying - two argument functions

```
def addA(x: Int, y: Int): Int = x + y

def addB(x: Int):Int=>Int = y => x + y

val a = addA(10, 20)

val b = addB(10)(20)

println(a)
println(b)
```

Currying - three argument functions

```
def addA(x: Int, y: Int, z: Int) = x + y + z

def addB(x: Int): Int => (Int => Int) =
y => (z => x + y + z)

val a = addA(1, 2, 3)

val b = addB(1)(2)(3)

println(a)
println(b)
```

It is now easy to see how the idea can be generalized to N argument functions!

Exercise

1. write a function that calculates the product of the values for the points on a gievn interval
2. write a function that calculates the product of the values of a function f for the points on a given interval
3. write the factorial in terms of product
4. can we write a more general function which generalizes both sum and product

6.3.1 Operazioni sulle collezioni - Map e Reduce

Immutability

Why? Immutable objects are automatically thread-safe (you don't have to worry about object being changed by another thread) Compiler can reason better about immutable values -> optimization

Steve Jenson from Twitter: "Start with immutability, then use mutability where you find appropriate."

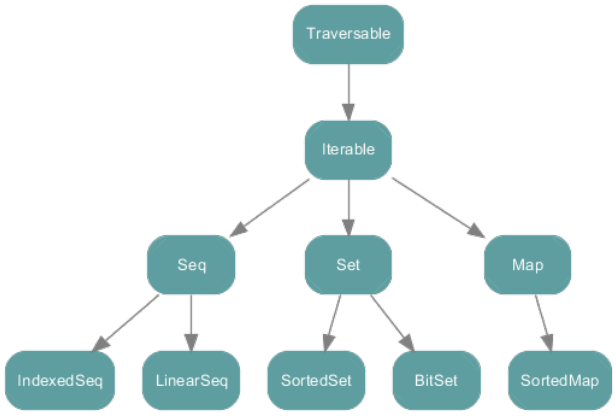
Collezioni

Mutable and Immutable Collections

- Scala collections systematically distinguish between mutable and immutable collections.
- A mutable collection can be updated or extended in place.
- Immutable collections, by contrast, never change.
- You have still operations that simulate additions, removals, or updates, but those operations will in each case return a new collection and leave the old collection unchanged.

Alto livello

collections in package scala.collection



Immutable

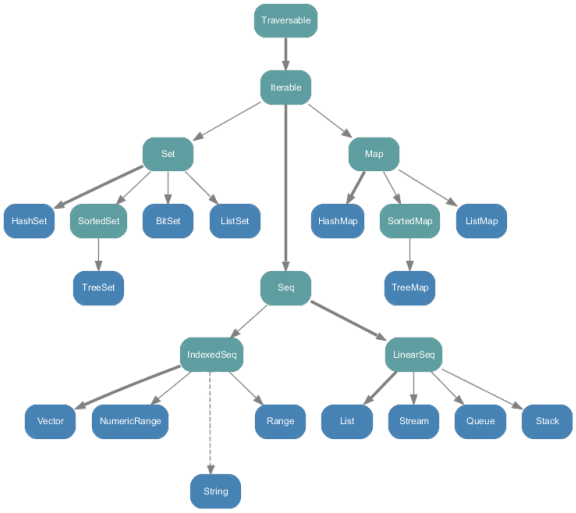
Queste sono le implementazioni immutabili

List

Le Lists sono immutabili (non può essere cambiato il contenuto)

List[String] contains Strings

```
val lst = List("b", "c", "d")
lst.head                    // "b"
lst.tail                    // List("c", "d")
val lst2 = "a" :: lst       // cons operator
```



Nil = synonym for empty list

```
val l = 1 :: 2 :: 3 :: Nil
```

List concatenation:

```
val l2 = List(1, 2, 3) ::: List(4, 5)
```

Foreach

Posso usare foreach per applicare una funzione a tutti gli elementi di una lista:

```
val list3 = List("mff", "cuni", "cz")
```

Following 3 calls are equivalent

```
list.foreach((s : String) => println(s))
list.foreach(s => println(s))
list.foreach(println)
```

For comprehensions

```
for (s <- list) println(s)
for (s <- list if s.length() == 4) println(s)
```

Maps and Sets

Ci sono anche mappe ed insiemi

Maps

Esempio di uso di mappe

```
import scala.collection._

val cache = new mutable.HashMap[String,String];
cache += "foo" -> "bar";

val c = cache("foo");
```

The rest of Map and Set interface looks as you would expect

Mutable List: ListBuffer

ListBuffer[T] is a mutable List Like Java's ArrayList<T>

```
import scala.collection.mutable._

val list = new ListBuffer[String]
list += "Vicky"
list += "Christina"

val str = list(0)
```

scala.Seq

- scala.Seq is the supertype that defines methods like: filter, fold, map, reduce, take, contains, ...
- List, Array, Maps... descend from Seq

From Java to Scala

```
Iterator              <=> java.util.Iterator
Iterator              <=> java.util.Enumeration
Iterable              <=> java.lang.Iterable
Iterable              <=> java.util.Collection
mutable.Buffer        <=> java.util.List
mutable.Set           <=> java.util.Set
mutable.Map           <=> java.util.Map
mutable.ConcurrentMap <=> java.util.concurrent.ConcurrentMap
```

Iterate – foreach function

Every collection in Scala's library defines (or inherits) a foreach method

```
val names = List("Daniel", "Chris", "Joseph")
names.foreach { name =>
        println(name)
}}
```

foreach is a "higher-order" method, due to the fact that it accepts a parameter which is itself another method

```
name => println(name)
names.foreach(println)
```

Foreach - istruzione

There are times that we just want to use a syntax which is similar to the for-loops available in other languages.

```
val nums = List(1, 2, 3, 4, 5)
var sum = 0
for (n <- nums) { sum += n }
```

Oppure se volessi usare una funzione + variabile locale:

```
var ss = 0;
def sinc(x:Int) = {      ss += x }
nums.foreach(sinc)
```

Folding

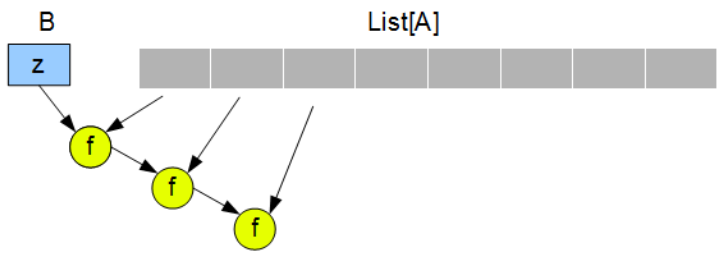
Looping is nice, but sometimes there are situations where it is necessary to somehow combine or examine every element in a collection, producing a single value as a result. Data una List[A]:

```
def foldLeft[B](z: B)(f: (B, A) => B): B
```

The foldLeft function goes through the whole List[A], from head to tail, and passes each value to f. For the first list item, that first parameter, z, is used as the first parameter to f. For the second list item, the result of the first call to f is used as the B type parameter.

foldLeft

```
def foldLeft[B](z: B)(f: (B, A) => B): B
```



Dove:

1. z è l'elemento iniziale di tipo B
2. f è l'operazione che dato un B e un A alla volta, costruisce mano a mano il dato riassuntivo di tipo B

Folding Esempi

1. Somma di tutti i numeri in nums

   ```
   def myf(x: Int, y: Int) = x+y
   val sum = nums.foldLeft(0)(myf)
   ```

2. oppure con funzione anonima

   ```
   val sum = nums.foldLeft(0)((total, n) => total + n)
   ```

Reduce

Fold has a closely related operation in Scala called "reduce" which can be extremely helpful in merging the elements of a sequence where leading or trailing values might be a problem. Consider the ever-popular example of transforming a list of String(s) into a single, comma-delimited value:

```
var nn = List("a","b", "c")// voglio stampare "a,b,c"
println(nn.foldLeft("")((x, y) => x + "," + y))
```

Stampa però: ,a,b,c

Reduce

Solution: use a reduce, rather than a fold. Reduce distinguishes itself from fold in that it does not require an initial value to "prime the sequence". Rather, it starts with the very first element in the sequence and moves on to the end.

```
def reduceLeft(f: (A, A) => A): A
```

Esempi:

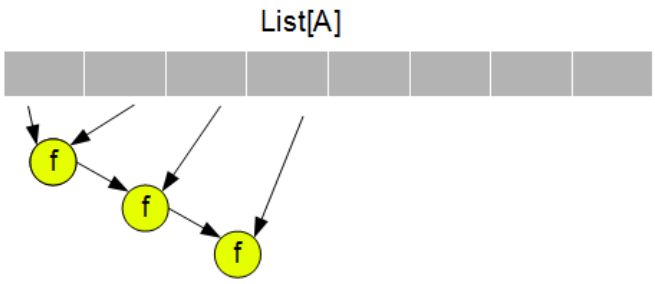
```
println(nn.reduceLeft((x, y) => x + "," + y))
--> a,b,c
```

Altro esempio: calcolo del max in lista

```
l.reduceLeft((x, y) => if (x>y) x else y )
```

Reduce

```
def reduceLeft(f: (A, A) => A): A
```



Esempi di Folder/Reduce

Fai la somma/prodotto dei numeri in una lista Restituisci la stringa piu' lunga Trova la dimensione della stringa piu' lunga ....

Filter/map

fold can be an extremely useful tool for applying a computation to each element in a collection and arriving at a single result if we want to apply a method to every element in a collection in-place (as it were), creating a new collection of the same type with the modified elements? Esempi, data una lista, costruire la lista dei doppi

```
var ll = List(3, 4, 5)
// lista dei doppi
def doppio(x: Int) = 2 * x
println(ll.map(doppio))
```

La lista delle lunghezze di una stringa

```
nomi.map( x => x.length())
```

Filter

Alcune volte voglio estrarre delle liste filtrando il contenuto Ad esempio: data una lista estrarre la lista pari

```
def pari = (x: Int) => (x % 2 == 0)
println(ll.filter(pari))
```

Using Map+Reduce

Spesso si usa map insieme a reduce:

- Con map trasformo i dati per renderli più trattabili
- Con reduce ottengo un dato sintetico
- Sono algoritmi che si possono parallelizzare
- Vedi google framework mapreduce http://it.wikipedia.org/wiki/MapReduce
- oppure spark http://spark.apache.org/