

Tipizzazione dei dati

Tipo (di dato)

- In un programma, ad ogni *entità* è associata informazione su:

- **Valore**: valore (attuale, calcolato, ...)
- **Tipo**: indicatore del tipo di dato associato
- **Dimensione**: dimensione dell'area di memoria occupata

Il *tipo di dato* ha conseguenze sull'**insieme di valori** che un'entità può assumere (valori ammissibili), la sua **dimensione in memoria** e le **operazioni** che su tali valori si possono effettuare

Definizione di tipizzazione

- Termine alternativo (inglese): *type checking*
- Informalmente: Processo che cerca di 'capire' qual è il **tipo** di un dato, sia esso dichiarato o prodotto/calcolato, in modo da poter effettuare i **relativi controlli**
 - vincoli su valori ammissibili
 - operazioni consentitecosì da poter garantire che il programma sia **type-safe**
- **Tutti i linguaggi di programmazione di alto livello** hanno un proprio sistema per la *tipizzazione* dei dati (ma con *significant differences...*)

Conseguenze della tipizzazione

- 1) Sicurezza
- 2) Ottimizzazione
- 3) Astrazione
- 4) Documentazione
- 5) Modularità

Conseguenze della tipizzazione: Sicurezza

- L'uso della tipizzazione permette di scoprire **codice privo di senso/illecito**
- **A livello di compilazione** → riduce i rischi di errori o risultati inattesi a run time
- Si consideri l'espressione **3 / “Hi World”**:
 - Operando 1: tipo semplice intero **3**
 - Operando 2: tipo stringa **“Hi World”**
 - Operatore: divisione **/**
- Es. Nel C l'operatore divisione **/** non prevede le stringhe → il compilatore emette un **semantic error**

Conseguenze della tipizzazione: Ottimizzazione

- Se eseguite in uno stadio iniziale (a livello di compilazione), le operazioni di type checking possono fornire informazioni utili per applicare tecniche di ottimizzazione sulla generazione del codice
 - Es. L'istruzione $x*2$ può essere ottimizzata nel seguente modo per produrre uno shift più efficiente:
 $mul\ x,2 \rightarrow shift_left\ x$
 - Possibile SOLO SE se si sa con certezza che x rappresenta un valore di tipo intero

Conseguenze della tipizzazione: Astrazione

- Il meccanismo dei tipi di dato permette di produrre programmi ad un **livello di astrazione più alto** di quello di linguaggi a basso livello
- Caso limite di assenza di tipizzazione: codice macchina nativo (sequenze di bit o byte – unico tipo)
- Es.: una stringa di caratteri è vista a basso livello come una **sequenza di byte**
 - Per l'essere umano, è molto più intuitivo pensare ad una stringa come ad una **concatenazione di caratteri** (es. tipo char)

Conseguenze della tipizzazione: Documentazione

- Nei sistemi di **tipizzazione più espressivi**, i nomi dei tipi di dato possono servire quale **fonte di documentazione del codice**
- Il **tipo di dato** illustra la natura di una variabile, ed in ultima analisi l'intento del programmatore
 - Es: **Booleani**, **Timestamp** o marcatore temporale (solitamente un intero a 32/64 bit)
 - Dal punto di vista della comprensione del codice, è diverso leggere **int a = 32;** o **timestamp a = 32;**
- Definizione di nuovi tipi di dato (**typedef** in C) per aumentare l'espressività del linguaggio

Conseguenze della tipizzazione: Modularità

- L'uso appropriato dei tipi di dato costituisce uno strumento semplice e robusto per definire **interfacce di programmazione (API)**
- Le funzionalità di una data API sono descritte dalle **signature** (i prototipi) delle funzioni pubbliche che la compongono
- Leggendo le signature, il programmatore si fa immediatamente un'idea di cosa può (o non può) fare
 - `boolean calcola(int a, int b, double c)`
 - `calcola(a,b,c)`

Type checking: quando?

- Le operazioni di **type checking** che **stabiliscono il tipo di dato associato a porzioni di codice e ne verificano i vincoli** (range di valori ammissibili e operazioni consentite) possono avvenire:
 1. **a tempo di compilazione (compile time)**
 2. **a tempo di esecuzione (run time)**

Classificazione non esclusiva: possibili anche vie intermedie

Type checking statico

- Il meccanismo di type checking è **statico** se le operazioni di type checking sono **eseguite solo a tempo di compilazione**
- Linguaggi a type checking statico:
 - C, C++, Fortran, Pascal, GO
- Il type checking statico:
 - permette di individuare parecchi **errori con largo anticipo** (inteso spesso come una **forma più sicura** di verifica di un programma)
 - permette **migliori prestazioni** (ottimizzazioni e **manca di controlli a run-time**)

Type checking dinamico

- Il type checking è **dinamico** se la **maggior parte** delle operazioni di type checking sono **eseguite a tempo di esecuzione**
 - *dichiarazione di tipo NON necessaria!*
- Linguaggi a type checking dinamico:
 - Javascript, Perl, PHP, Python, Ruby
- **Più flessibile** di quello statico, con maggior overhead
 - **Le variabili possono cambiare tipo a run time**
 - **Strutture dati di forma mutevole a run time:** il tipo di dato può cambiare e va quindi “gestito”
 - Pensate ad un array (valori contigui in memoria...)

Type checking dinamico: assegnamenti da input

- Godendo di un meccanismo di tipizzazione a run time, si possono effettuare assegnamenti “arditi”
- Ad esempio: `var = <token letto da input>;`
dove token può avere un qualunque tipo (intero, stringa)
- Il type checker dinamico assegna il **tipo corretto a run time**
- *Pensate a scrivere una cosa simile in C o in Java ...*

Prova - Perl

→ESEMPI
perl/provaTipi.pl

```
#!/usr/bin/perl
$var=<STDIN>; #lettura da input
print "$var";
$sum=$var+56;
print "Sum $sum\n";
$div=$var/6;
print "Div $div\n";
```

```
#input 12
12
Sum 68
Div 2
```

```
#input 3.5
3.5
Sum 59.5
Div 0.58333333333333
```

```
#input ciao
ciao
Sum 56
Div 0
```

```
#input "12"
"12"
Sum 56
Div 0
```

```
#input 12ciao
12ciao
Sum 68
Div 2
```

Type checking dinamico: minori garanzie

- *Ovviamente, la flessibilità si paga...*
- Un type checker dinamico dà **minori garanzie a priori** perché opera (per la maggior parte) a run time
- Esempio precedente: supponiamo \$var intesa come variabile intera - se a run time viene fornita una stringa, l'interprete non potrà fare altro che **non considerarla, tentare di convertirla o generare un errore**
 - *Maggiore rischio di risultato inaspettato*

Type checking dinamico: necessità di test accurati

Possibilità di incorrere in tante situazioni di
“errore” causa input inaspettati

- Eterogeneità delle reazioni a run time a seconda del linguaggio

Maggiore necessità di:

- Gestire le situazioni di errore a run time - con meccanismi di gestione delle eccezioni
- Verificare la correttezza di un programma - con test completi e accurati

Type checking ibrido

- Alcuni linguaggi fanno uso **ibrido** di tecniche di type checking statico e dinamico
- **Java:**
 - Checking statico (a compile time) dei tipi
 - es. “2”/70 dà errore in compilazione
 - Checking dinamico (a run time) per le operazioni di **binding tra metodi e classi** (legato al **polimorfismo**: identificazione di una signature valida nella gerarchia delle classi)

NOTA: Java si considera comunque un linguaggio a type checking statico

Vantaggi a confronto

- **Type checking statico:**
 - Identifica errori a tempo di compilazione (più conservativo – *necessita di meno controlli*)
 - Permette la costruzione di codice che esegue più velocemente
- **Type checking dinamico:**
 - Permette una più rapida prototipazione
 - È più flessibile: permette l'uso di costrutti considerati illegali nei linguaggi statici
 - ES: `var x; x=5; x="ciao"; var=<input>`
 - Permette la generazione di nuovo codice a runtime (metaprogramming – es. funzione `eval()`)

Tipizzazione forte

- Un linguaggio di programmazione adotta una **tipizzazione forte** se **impone regole rigide e impedisce usi incoerenti** dei tipi di dato specificati (es. operazioni effettuate con operandi aventi tipo di dato non corretto)
- ES.
 - Un'operazione di somma di interi con caratteri
 - Un'operazione in cui l'indice supera i limiti della dimensione di un array
- **Non esiste un linguaggio completamente tipizzato in maniera forte**
 - *Sarebbe quasi inutilizzabile...*

Tipizzazione debole

- Un linguaggio di programmazione adotta una **tipizzazione debole** dei dati se **non impedisce operazioni incongruenti** (es. con operandi aventi tipo di dato non corretto)
- La tipizzazione debole fa spesso uso di operatori di conversione (**casting implicito**) per rendere **omogenei gli operandi**
 - Spesso usati in C:
 - Si considerino le espressioni **'a' / 5** e **30 + "2"**
 - Java è fortemente tipizzato (più del C e del Pascal)
 - Perl ha una tipizzazione molto debole

Tipizzazione debole

- ATT: con la tipizzazione debole il *risultato può cambiare a seconda del linguaggio*
- Esempio: `var x:= 5; var y:= “37”; y + x;`
- *Quante possibili interpretazioni possono esserci?*

Tipizzazione debole

- ATT: con la tipizzazione debole il *risultato può cambiare a seconda del linguaggio*
- Esempio: `var x:= 5; var y:= "37"; y + x;`
 - In C: aritmetica dei puntatori
 - In linguaggi Java-based (Javascript, Java), x viene convertito a stringa (`x+y="537"`)
 - In Visual Basic e in Perl, y viene convertito ad intero (`x+y=42`)
 - In Perl una stringa non contenente cifre iniziali (es. `"ciao37"`) viene considerata nulla
 - In Python non viene accettato: esce con errore (*anche i linguaggi dinamici possono talvolta essere intransigenti...*)

Tipizzazione safe

- Un linguaggio di programmazione adotta una **tipizzazione safe** (sicura) dei dati se non permette ad una operazione di casting implicito di produrre un crash
- Esempio (Perl):
`$x = 5; $y = "37"; $z = $x + $y;`
- In questo esempio, il risultato è 42
- L'operazione di conversione non fallisce su una stringa contenente un intero
- Con stringa "Hi world", la conversione di \$y produrrebbe il valore 0 e a \$z verrebbe assegnato il valore 5 (sempre senza crash)

Tipizzazione unsafe

- Un linguaggio di programmazione adotta una **tipizzazione unsafe** (non sicura) dei dati se consente operazioni di casting che possono produrre un crash
- Esempio (C):
`int x= 5; char y[] = "37"; char *z = y + x;`
 - z viene fatto puntare all'indirizzo di memoria 5 byte più in avanti di y
 - Il contenuto di z non è definito, e può trovarsi al di fuori dello spazio di indirizzamento del processo
 - Una dereferenziazione di z può portare al crash (segmentation fault)

Esempio Perl

```
#!/usr/bin/perl
@num=(1,2,3);
print "@num\n";
$num[0]=4;
print "@num\n";
print $num[5];
$num[5]=4;
print "@num\n";
```

Esempio Perl

```
#!/usr/bin/perl
@num=(1,2,3);
print "@num\n";
$num[0]=4;
print "@num\n";
print $num[5];
$num[5]=4;
print "@num\n";
```



```
1 2 3
4 2 3
4 2 3 4
```

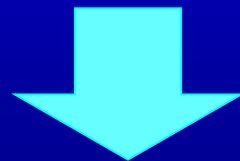
Duck typing

Tipizzazione e polimorfismo

- **Polimorfismo** = capacità di differenziare il comportamento di parti di codice in base all'entità a cui sono applicati → *riuso del codice*
- Nello schema classico (es. Java) di un linguaggio ad oggetti, il **polimorfismo** è solitamente legato al meccanismo di **ereditarietà**
- L'ereditarietà garantisce che le classi possano avere una **stessa interfaccia**:
 - ES: in Java le istanze di una sottoclasse possono essere utilizzate al posto di istanze della superclasse (**polimorfismo per inclusione**)
 - L'**overriding** dei metodi permette che gli oggetti appartenenti alle sottoclassi rispondano diversamente agli stessi utilizzi (**metodi polimorfi**)

Polimorfismo classico ad oggetti

- In questo caso, **applicare il polimorfismo e individuare la signature valida di un metodo** implica:
 - Individuare la classe di appartenenza dell'oggetto, e cercarvi una signature valida per il metodo
 - In caso di mancata individuazione, ripetere il controllo per tutte le superclassi (**operation dispatching** – potenzialmente risalendo fino alla classe radice, es. **Object**)



Implica risalire la gerarchia di ereditarietà e cercare la prima “classe giusta” (una con la signature valida per il metodo)

Polimorfismo classico ad oggetti

- La ricerca della classe adatta nella gerarchia delle classi può essere effettuata
 - *A tempo di compilazione*
 - *A tempo di esecuzione*
- In molti linguaggi a oggetti (es. C++) avviene a tempo di compilazione (*il costo a tempo di esecuzione è ritenuto troppo elevato*)
- In Java e nei linguaggi dinamici, avviene a tempo di esecuzione → può essere molto costoso in termini di overhead

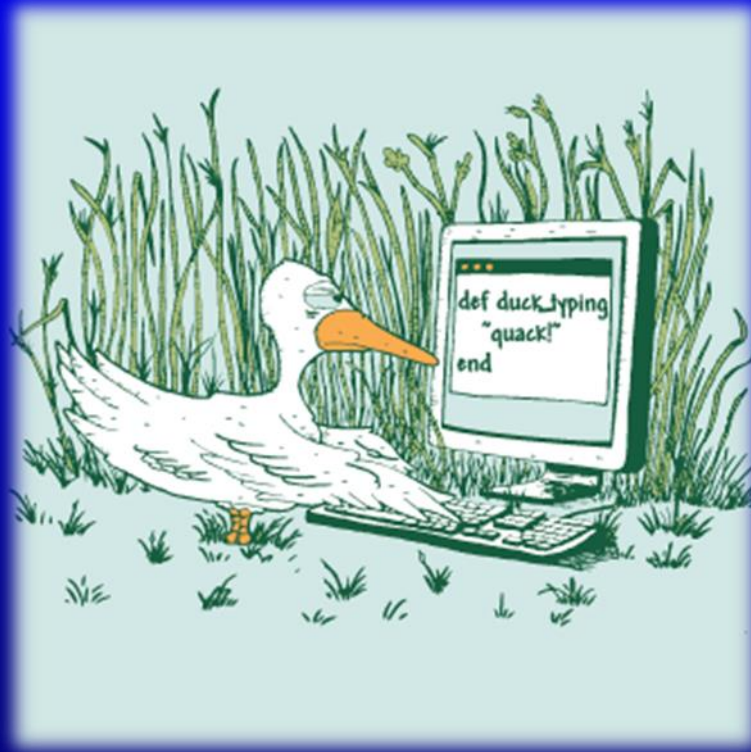
Ma nei linguaggi dinamici c'è una alternativa...

Duck typing

- Nei linguaggi dinamici (ad oggetti) → meccanismo detto **duck typing**
- Il **duck typing** permette di realizzare il concetto di **polimorfismo** senza dover necessariamente usare meccanismi di ereditarietà (o di implementazione di interfacce condivise)
- *“Se istanzio un oggetto di una classe e ne invoco metodi/attributi, l'unica cosa che conta è che **i metodi/attributi siano definiti per quella classe**”*
- Meccanismo possibile grazie alla presenza di type checking dinamico: il controllo (**duck test**) viene effettuato dall'interprete a **run time**

Duck test

"When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck"



Newsgroup di
Python nel 2000

Duck typing

- Esempio: Animale, Cane e Gatto
 - *Una funzione funz()* prende in ingresso un **parametro formale (non tipizzato!)** e ne invoca il metodo *CosaMangia()*
 - **non è necessario** che le classi *Cane* e *Gatto* siano sottoclassi di *Animale* per essere passate a *funz()* ed utilizzate: basta che abbiano un metodo *CosaMangia()*
- All'interprete interessa solo che i tipi di oggetto **espongano un metodo con lo stesso nome e con lo stesso numero di parametri in ingresso**

Esempio (in Python)

```
class Duck:  
    def quack(self):  
        print("Quaaaaaack!")
```

```
class Person:  
    def quack(self):  
        print("The person imitates a duck.")
```

```
def in_the_farm(a):  
    a.quack()
```

Classi senza relazione di parentela, né interfacce comuni, ma aventi un metodo con stesso nome e numero di parametri

Funzione che invoca un parametro a (non tipizzato, type checking dinamico!) e ne invoca il metodo quack()

Esempio (in Python)

```
class Duck:  
    def quack(self):  
        print("Quaaaaaack!")
```

```
class Person:  
    def quack(self):  
        print("The person imitates a duck.")
```

```
def in_the_farm(a):  
    a.quack()
```

```
def game():  
    donald = Duck()  
    john = Person()  
    in_the_farm(donald)  
    in_the_farm(john)
```

A run time l'interprete controlla
che il metodo quack() sia
implementato sia in Duck
che in Person (duck test)

Tipizzazione e polimorfismo

- In generale, il **polimorfismo a livello di tipi di dato** permette di:
 - **differenziare il comportamento dei metodi** in funzione del tipo di dato a cui sono applicati
 - evitare di dover predefinire un metodo, classe, o struttura dati appositi per ogni possibile combinazione di tipi di dati→ **Riuso del codice**
- **NOTA: in un linguaggio tipizzato dinamicamente, tutte le espressioni sono intrinsecamente polimorfe**
 - *calcola(a,b,c)*

Esempio

- Creazione di funzioni che prendono un oggetto di qualsiasi tipo e ne usano i metodi (purché definiti)

Pseudo codice per linguaggio dinamico

```
function calcola(a,b,c) => return (a+b)*c
```

```
e1 = calcola(1,2,3)
```

```
e2 = calcola([1,2,3],[4,5,6],2)
```

```
e3 = calcola('mele ', 'e arance', 3)
```

Esempio

- Creazione di funzioni che prendono un oggetto di qualsiasi tipo e ne usano i metodi (purché definiti)

Pseudo codice per linguaggio dinamico

```
function calcola(a,b,c) => return (a+b)*c
```

```
e1 = calcola(1,2,3)
```

```
e2 = calcola([1,2,3],[4,5,6],2)
```

```
e3 = calcola('mele ', 'e arance', 3)
```

Se tradotto in Ruby o Python:

e1 → 9

e2 → [1, 2, 3, 4, 5, 6, 1, 2, 3, 4, 5, 6]

e3 → mele e arance, mele e arance, mele e arance

**Funziona fintanto che
gli oggetti supportano i
metodi “+” e “*”**

Concludendo

- Quasi tutti i **linguaggi dinamici** adottano un meccanismo di gestione degli attributi e dei metodi degli oggetti basato su **duck typing**
 - Perl, PHP, Ruby, Python
- Permette di **ottenere polimorfismo senza dover per forza incorrere nell'overhead dovuto all'ereditarietà**
- *Ovviamente anche qui si deve considerare la possibilità di errori a tempo di esecuzione:*
 - in questo modo non è possibile imporre che gli oggetti rispettino una interfaccia comune

Esempio

- **Confronto Java vs Python**
- **Due classi che rappresentano una forma geometrica ed hanno una interfaccia comune metodo draw() che stampa le coordinate principali della forma**
 - **Circle**
 - **Square**
- **Painter definisce una lista di 4 oggetti – 2 circle e 2 square, su cui invoca il metodo draw()**

Alternative per la programmazione generica

→ESEMPI
stack.cpp
stack.py

- Anche **alcuni linguaggi tipizzati staticamente** possono fare uso di un meccanismo alternativo al duck typing per permettere la **programmazione generica**
 - Il C++ e Java utilizzano **Template** e **Generics** come meccanismi di supporto alla **programmazione generica**, per scrivere codice funzionante indipendentemente dal tipo di dato che verrà fornito
 - Es. stack.cpp realizza una **classe template** con un **parametro generico T**, che sarà poi sostituito a tempo di compilazione da diversi tipi di dato (int,float)
 - Gestione stack con funzioni push(T), pop(), isEmpty(), isFull()