**Emerging Programming Paradigms**

[ lezioni 06 maggio ]

In OCaml gli oggetti permettono il **dynamic dispatch** e il **polimorfismo ad-hoc** (in collaborazione con gli **object types** (quelle righe in OCaml tipate con polimorfismo di riga) che assumono il ruolo di **interfacce**).

Le classi hanno invece a che fare con l’**ereditarietà** che è un meccanismo di riuso del codice che crea vari problemi, tra cui confusione con il tipaggio (inheritance non è subtyping) e mondo aperto (non si sa a compile time che codice verrà eseguito e quindi come garantire gli invarianti).

Inheritance si riferisce al riuso di implementazioni.

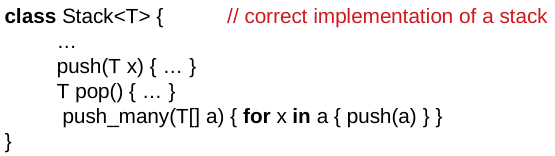
Subtyping si riferisce alla compatibilità di interfacce.

Inoltre l’inheritance richiede la catalogazione in rigide gerarchie arborescenti che descrivono cosa un oggetto è e non cosa fa. Le *interfacce* invece ci dicono cosa un oggetto fa e sono più rilevanti delle classi dal punto di vista del tipaggio (non sono infatti un meccanismo di riuso del codice).

Cambiare l’interfaccia di una classe rompe tutto il codice che usa classi derivate. Una sottoclasse deve implementare tutto il behaviour della superclasse anche se non necessario, uguale per l’istanziazione di campi. Ad esempio data la classe Lista se si volesse implementare uno Stack indicandolo come una Lista con un pop e un push, bisognerebbe andare a reimplementare (riuso del codice) tutti metodi e campi forniti da lista, quando molti non servono. In questo modo si sta quindi esponendo a Stack il behaviour di Lista, che non andrebbe esposto, inoltre se cambiasse l’interfaccia di Lista si andrebbe a rompere anche Stack in quanto classe derivata.

Quando una sottoclasse rompe gli invarianti della superclasse aggiustare il codice può essere molto costoso (refactoring della gerarchia).

Data la seguente classe:



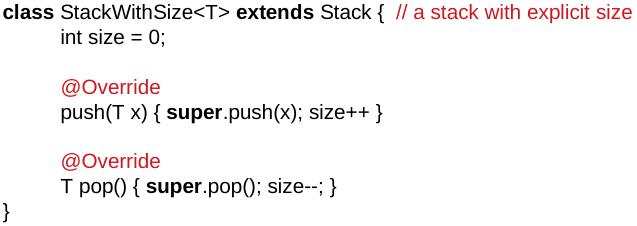
Questa è un’implementazione corretta dello stack, perché se si esplicitano gli invarianti e precondizione e postcondizione dei metodi tutto rimane corretto. In questo alcuni *esempi di invarianti* possono essere:

- se lo stack è vuoto e viene fatto pop viene lanciata un’eccezione

- se si fa una push e una pop si ottiene l’elemento pushato

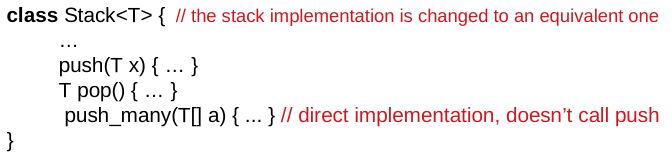
- …

Data la classe StackWithSize che eredita da Stack e gli da una dimensione esplicita:



La push\_many non è implementata perché quando verrà istanziata StackWithSize verrà chiamata la push\_many di Stack in quanto senza override, la quale chiamerà la push di cui è stato fatto l’override.

Supponiamo ora di implementare diversamente la classe Stack:



In questa implementazione, del tutto equivalente alla precedente, la push\_many *non* utilizza la push, ma ad esempio concatena l’array a alla lista.

Tuttavia se StackWithSize rimane la stessa il codice non funzionerebbe più in quanto, non chiamando più la push, non verrebbe incrementata la variabile size. Se si pensasse di fare l’override della push\_many per aggiustare il problema andrebbe bene, ma nel caso in cui Stack venisse cambiata nuovamente come prima si romperebbe di nuovo il codice.

Questo problema è chiamato **fragile base class problem**.

**Composition**

Una soluzione proposta per questo tipo di problemi è la *Composition*. Ovvero un oggetto di una classe B non eredita dalla classe A, ma ha un campo che è un oggetto di una classe A. In questo modo B può esporre i metodi della classe A selettivamente, quindi può **evitare di esporre** forzatamente **tutto il behaviour di A**. Nell’esempio di prima StackWithSize avrebbe due campi, size e stack, i metodi push e pop chiamerebbero quelli di stack e reimplementando push\_many verrebbero chiamati i relativi metodi di Stack, non quelli di cui si è fatto l’override, quindi resisterebbe a eventuali modifiche interne di push\_many di Stack.

Non vengono inoltre imposte gerarchie e non si hanno problemi di sub-typing.

Se si tolgono le classi, non si hanno oggetti, e se non si hanno oggetti si possono creare metodi senza oggetti (**object-less methods**).

**Go**

Go è un linguaggio recente, adatto allo sviluppo di microservizi. Si discosta poco da C, ha *garbage collection* e un sistema di tipi debole, stile anni ‘60. Go ha funzioni e tipi di dato user-defined del tutto analoghi al C.

Go ha metodi senza avere oggetti, un metodo altro non è che una funzione a tutti gli effetti (anche nella rappresentazione a runtime) per la quale viene dichiarato un receiver di un tipo dichiarato nel file corrente. I metodi vengono invocati con la sintassi o.m(t1, …, tn) invece che con la tradizionale m(o, t1, …, tn). Questo zucchero sintattico è molto utile per l’autocompletion degli IDE.

Questi non sono pre-metodi, in quanto un *pre-metodo* sarebbe o.m(o, t1, …, tn), qui si parla invece di **metodi liberi**, nel senso delle chiusure. I metodi qui non sono “ingabbiati” in un oggetto, il metodo prende in input il self (ovvero o) ma il metodo non viene estratto da o, bensì è “libero” ed essendo libero è possibile **invocarlo staticamente** e non dinamicamente (estraendolo la o).

Esempio:

|  |  |
| --- | --- |
| func Abs(v Vertex) float64 {  return math.Sqrt(v.X\*v.X + v.Y\*v.Y)  }  func main() {  v := Vertex{3, 4}  fmt.Println(Abs(v))  } | func (v Vertex) Abs() float64 {  return math.Sqrt(v.X\*v.X + v.Y\*v.Y)  }  func main() {  v := Vertex{3, 4}  fmt.Println(v.Abs())  } |

Nella versione a sinistra la funzione prende in input un Vertex e l’invocazione richiederà che venga passato come argomento un Vertex.

Nella versione a destra invece la funzione dichiara prima un **receiver** Vertex, in questo modo l’input viene specificato prima della chiamata di funzione. In questo modo viene dichiarato un nuovo metodo Abs nella struttura Vertex, senza doverglielo specificare nella dichiarazione all’inizio. Questa operazione può essere fatta anche su puntatori non solo su strutture.

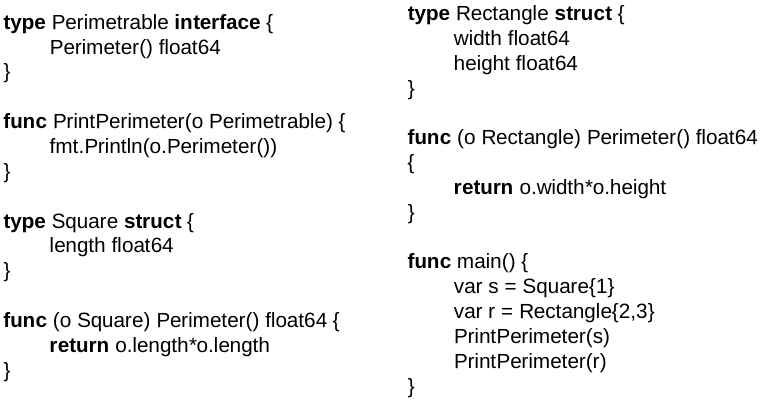
Un “metodo” di Go non è altro che una funzione invocata con una sintassi differente. Non ha tipaggio speciale, la funzione non è un campo di un record, non viene estratta da un record e l’invocazione è molto efficiente in quanto basata di *static dispatch*.

NOTA: mentre in OCaml il connettivo principale era un “per ogni” in Go il connettivo è l’”esiste”, di fatto si sta quantificando con un per ogni in posizione di argomento, e i per ogni in posizione di argomento sono degli esiste.

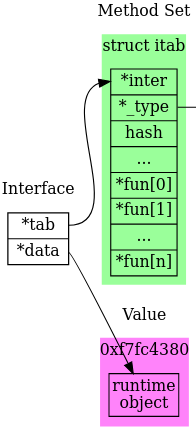
**Interfacce**

Per avere **polimorfismo di riga** Go introduce la nozione di **interfaccia**, un insieme di method signatures (ovvero una riga). Viene usata per tipare gli argomenti delle funzioni polimorfe (così come gli object type di OCaml). Le funzioni che prendono argomenti di tipo interfaccia vengono invocate con *dynamic dispatch*, questo perché se si passa a una funzione un argomento di tipo interfaccia il compilatore non sa qual è il suo tipo concreto e non può staticamente decidere quale metodo verrà invocato. Per implementare un’interfaccia semplicemente si implementano tutti i metodi, duck typing: se per essere una papera bisogna nuotare e fare quack, è sufficiente implementare (almeno) questi metodi.

Esempio di utilizzo:



Quindi viene implementata la funzione Perimeter, dichiarata da Perimetrable su ogni tipo in modo indipendente e a se stante.



Un valore di tipo interfaccia (ad esempio “o Perimetrable” in PrintPerimeter) a runtime ha due possibilità, può essere nil (in quanto non ha polimorfismo uniforme, e non ha quindi options), oppure una coppia (v, p), in cui v è il valore che viene passato (che può essere un valore numerico tipo 5 oppure un puntatore ad un dato nello heap), mentre p è un puntatore alla tabella dei metodi per quell’interfaccia.

Prendendo l’esempio sopra, PrintPerimeter lavora su un’interfaccia Perimetrable che deve implementare Perimeter(). Nel momento in cui il compilatore vede nel main PrintPerimeter(s), capisce che s è un quadrato e costruisce a compile time un record in memoria che avrà tutti i metodi dell’interfaccia Perimetrable (in questo caso solo Perimeter()) implementati per il quadrato.

Pensando ai **linguaggi a oggetti** verrebbe passato l’oggetto a PrintPerimetrable, il quale sarebbe un **record** con tanti **campi** e un **puntatore** alla **tabella dei metodi**. In questo caso vengono passati i **dati** (length per Square o width, height per Rectangle) e il **puntatore** alla **tabella dei metodi** esplicitamente come due dati separati (v, p).

NOTA: esiste anche la coppia (nil, p) che è valida e permette di passare nil a una funzione o a un metodo. Significa se si sta invocando codice su un puntatore nullo ma su cui i metodi sono definiti (ad esempio nil per la lista vuota e p una funzione che lavora su liste e sa gestire anche il caso nil).

**Cast**

Se o ha come tipo un’interfaccia I, allora:

- x := o.(T) allora si sta assegnando ad x il valore di o (la prima proiezione di o) se e solo se o ha tipo T (la seconda proiezione punta a T). In sostanza si sta prendendo un o di interfaccia I e lo si sta castando a T, e se ha successo si recupera il puntatore x.

- x, err := o.(T) è uguale a prima ma prevede la possibilità di fallimento. Se il cast non ha successo x è nil e err è true. Non avendo gli ADT non è possibile fare l’”or” in un tipo di dato, ma si usa l’”and”, quindi x è un valore (nil in questo caso) AND err è true.

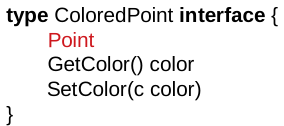
È possibile fare cast anche con lo switch, controllando i.(type) quale tipo abbia.

Il fatto di non avere polimorfismo uniforme (generics, templates, …) fa si che in Go tutti i container (alberi, stack, …) contengano valori di tipo **interface {}** (interfaccia vuota) perché a volte può essere richiesto un intero altre volte un float, ecc e non si sa quale interfaccia usare, quindi si mette vuota.

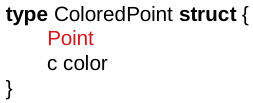
L’assenza di tipi di dati algebrici indica l’assenza di unioni disgiunte di tipi, quindi non si hanno valori taggati.

**Inheritance**

Go non ha inheritance ma ha **Composition**. Ad esempio:



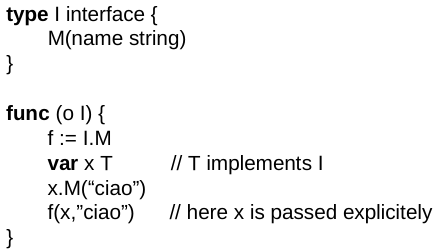
Un’interfaccia può includere tutti i metodi di un’altra interfaccia tramite il nome della seconda.



Scrivendo semplicemente il nome della struttura Point, viene fatto composition, quindi a runtime ColoredPoint ha effettivamente un punto, ma automaticamente per ogni metodo di Point viene generato il medesimo per ColoredPoint che richiama il proprio omonimo in Point.

L’**override non avviene**, perché non si sta facendo inheritance e i metodi di Point venendo chiamati su Point lavoreranno comunque su di esso anche se chiamati su ColoredPoint.

È possibile estrarre un metodo da un’interfaccia assegnandolo ad una funzione:



o è un’interfaccia quindi a runtime una coppia (v, p) e con I.M viene detto di prendere il secondo valore p (puntatore a tabella dei metodi) e di estrarre M.

È impossibile definire un metodo il cui receiver sia di tipo interfaccia, questo perché sarebbe impossibile fare static dispatch, il compilatore non saprebbe associare del codice all’interfaccia, dovrebbe andare in tutte le tabelle dei simboli di tutti i tipi che implementano l’interfaccia, e ciò non ha senso.

È possibile anche dichiarare un metodo che opera su tipi funzioni, per esempio decorandole (questo è possibile perché è un linguaggio higher order e si possono prendere in input funzioni).

Go è un linguaggio di basso livello ma non è un linguaggio di sistema. Go ha un sistema di runtime molto pesante che implementa tra le altre cose anche un su garbage collection, quindi non può essere usato per codice embedded o parte di un sistema operativo. A differenza di Rust che è un linguaggio di basso livello e di sistema.