PROGRAMMAZIONE ORIENTATA AGLI OGGETTI

Object-Oriented Programming (OOP)

Modularità dei programmi

I linguaggi di programmazione supportano in vari modi la possibilità di usare la modularizzazione dei programmi

- **funzionale**: sotto il profilo linguistico, con l'astrazione procedurale (possibilità di scomporre il problema in sotto-problemi da risolvere con specifiche procedure/funzioni, aiutando a mantenere la l'attenzione sul flusso logico)

```
int main() {
:
int x = sotto_problema();
:
}

int sotto_problema() { ... }
```

Modularità dei programmi

- **per tipo**: sotto il profilo dei tipi di dato, con i tipi di dato astratti

```
module type BOOL = sig
  type t
  val yes: t
  val no: t
  val choose: t -> 'a -> 'a -> 'a
```

Il **modulo** definisce un **tipo astratto** che rappresentante il **tipo booleano**, con

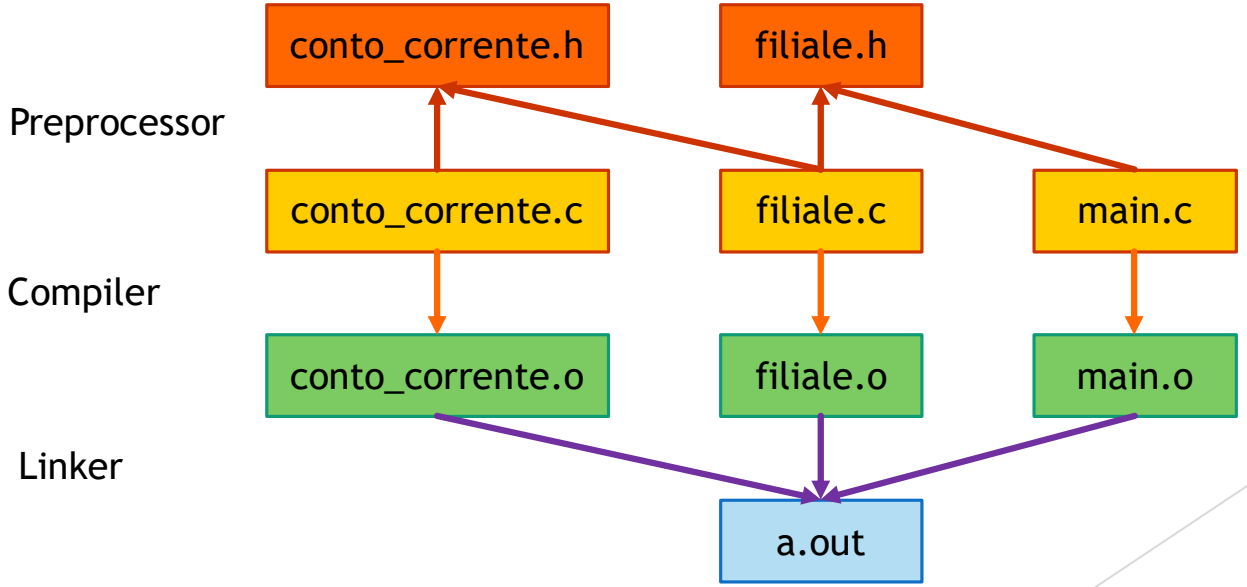
- due valori (vero e falso), e
- una funzione che, in base al primo valore booleano, restituisce uno degli altri due valori

```
module M1 : BOOL = struct
  type t = unit option
  let yes = Some ()
  let no = None
  let choose v ifyes ifno =
    match v with
    | Some () -> ifyes
    | None -> ifno
end
```

Istanza del modulo BOOL, chiamata M1, che implementa: un tipo booleano utilizzando il tipo unit option, e la funzione choose

Modularità dei programmi

- **Organizzazione compilazione**: sotto il profilo delle tecniche di compilazione ed esecuzione, con la compilazione separata (che consente di compilare i moduli in modo indipendente) e il linking (che unisce i vari file oggetto generati dalla compilazione separata in un unico eseguibile)



Livelli di astrazione

La **modularizzazione** dei programmi consente di progettare e sviluppare un programma per **livelli di astrazione**

Ad esempio:

- la libreria standard consente di scrivere programmi che operano su stringhe astraendo da (cioè, ignorando) come le operazioni su stringhe siano implementate
- per implementare un programma di gestione di banche:
  1. si parte implementando i moduli di gestione dei singoli conti correnti
  2. si passa a implementare i moduli di gestione di una filiale, usando i conti correnti, ma astraendo dalla loro implementazione
  3. infine si implementano i moduli di gestione della rete di filiali usando il modulo della singola filiale, ma astraendo dalla sua implementazione

Livelli di astrazione

Tutti i sistemi informatici complessi sono organizzati per livelli di astrazione. Ad esempio:

- linguaggi di programmazione

  Codice sorgente -> Bytecode -> Assembler

- sistemi operativi

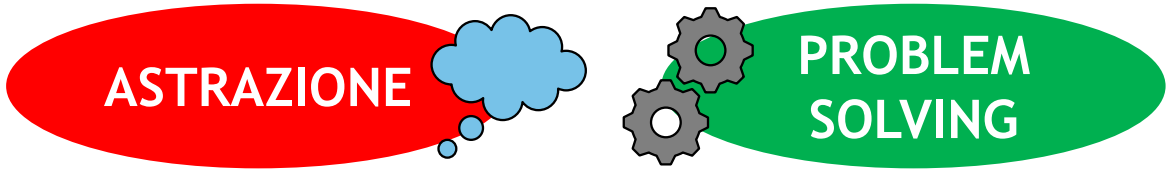
  Applicazione -> Sistema Operativo -> Hardware

- protocolli di comunicazione su reti

  HTTP -> TCP -> IP -> Ethernet

Astrazione + problem solving

Le capacità di



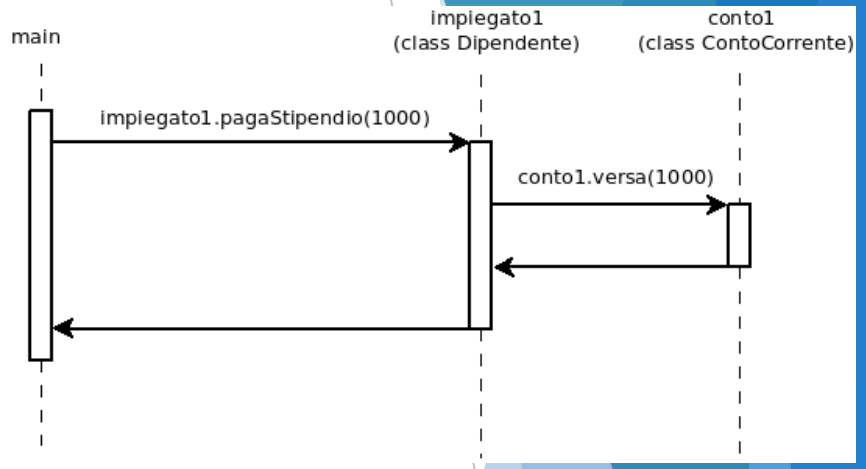
caratterizzano i buoni informatici

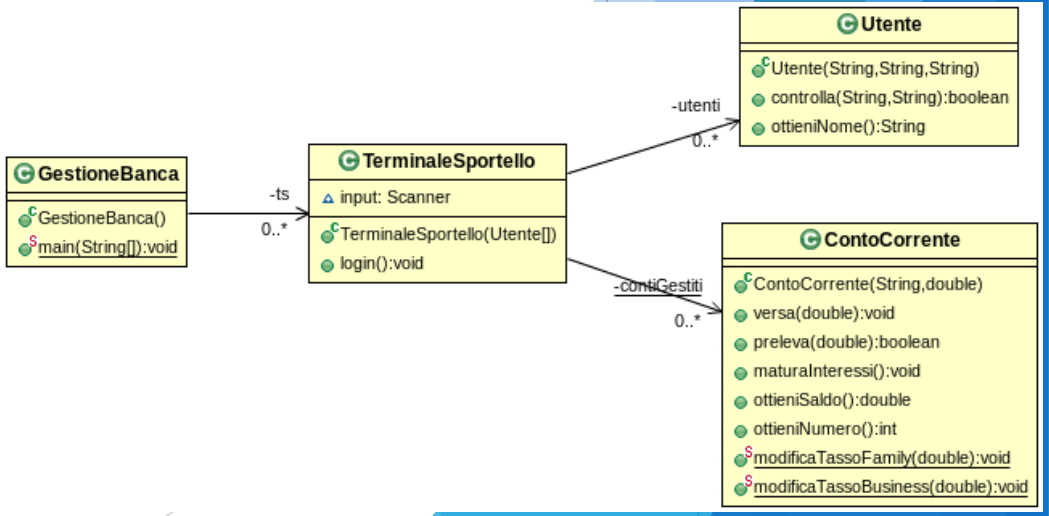
- l'**astrazione** permette di semplificare problemi complessi, focalizzandosi sugli aspetti essenziali e ignorando i dettagli superflui
- il **problem solving** consente di analizzare i problemi, scomporli in parti risolvibili e trovare soluzioni efficienti

Modularità e Ingegneria del Software

Sviluppare un programma complesso in modo modulare consente inoltre di suddividere il lavoro tra sviluppatori diversi

- Si definiscono le specifiche (e le interfacce) delle diverse parti
- Ogni sviluppatore lavora sulla propria parte seguendo le specifiche e assumendo che anche gli altri le seguano
- Esistono metodi di Ingegneria del Software per definire le specifiche in modo non ambiguo, utilizzando strumenti standard come i diagrammi UML

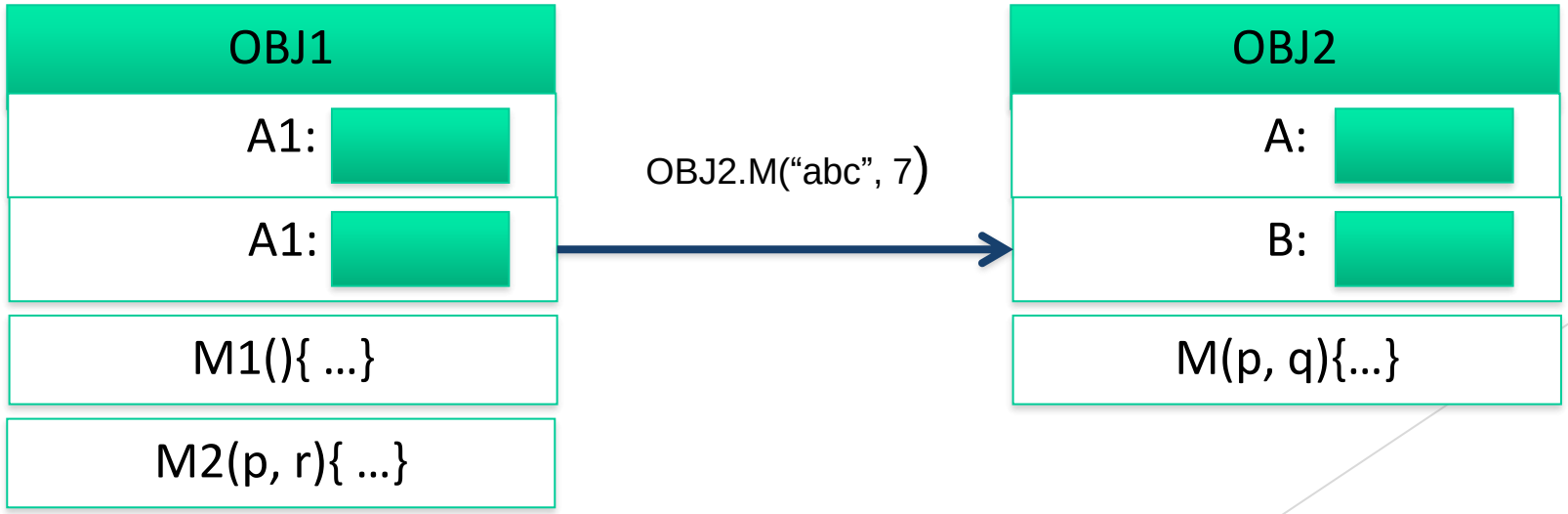




IL PARADIGMA OBJECT-ORIENTED

Il paradigma a oggetti

- Sistema software = insieme di oggetti che cooperano tra loro
- Gli oggetti sono caratterizzati da:
  - uno STATO, che rappresenta la struttura interna dell'oggetto
  - un insieme di FUNZIONALITÀ (o metodi) , che definiscono le operazioni che l'oggetto può eseguire
- Gli oggetti cooperano scambiandosi messaggi

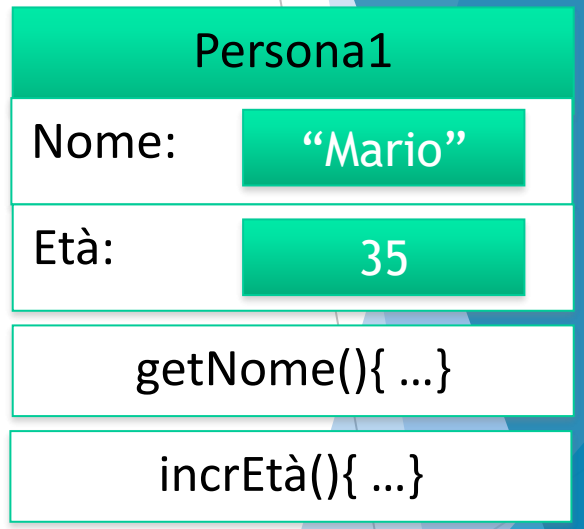


Lo STATO di un oggetto

Lo STATO di un oggetto è solitamente rappresentato da un insieme di attributi/proprietà/variabili

- Proprietà di INCAPSULAMENTO

Idealmente, lo stato di un oggetto non dovrebbe essere accessibile agli altri oggetti



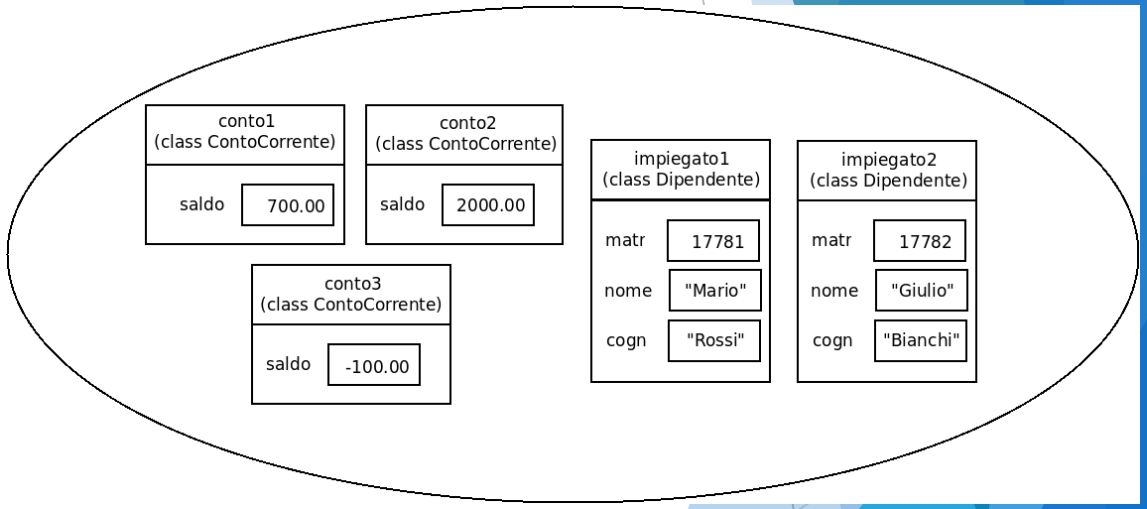
Un oggetto A non dovrebbe poter leggere/modificare direttamente le variabili che rappresentano lo stato di un altro oggetto B (INFORMATION HIDING)

- Anzi, A non dovrebbe nemmeno aver bisogno di sapere come lo stato di B sia rappresentato all'interno (cioè, che variabili usa) …

Lo stato del programma

Idealmente, in un linguaggio basato solo sul paradigma object-oriented, lo stato del programma corrisponde all'insieme degli stati degli oggetti che lo compongono

- Gli **oggetti** sono responsabili di mantenere il proprio stato interno, e l'interazione tra oggetti modifica lo stato complessivo del programma
- In aggiunta a questo, ci saranno le strutture dati di sistema necessarie per l'esecuzione del **supporto a run-time** (ad es. il run-time stack)

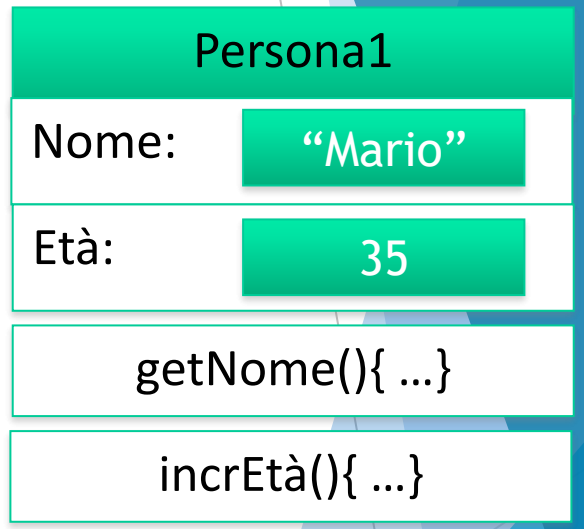


Le FUNZIONALITÀ di un oggetto

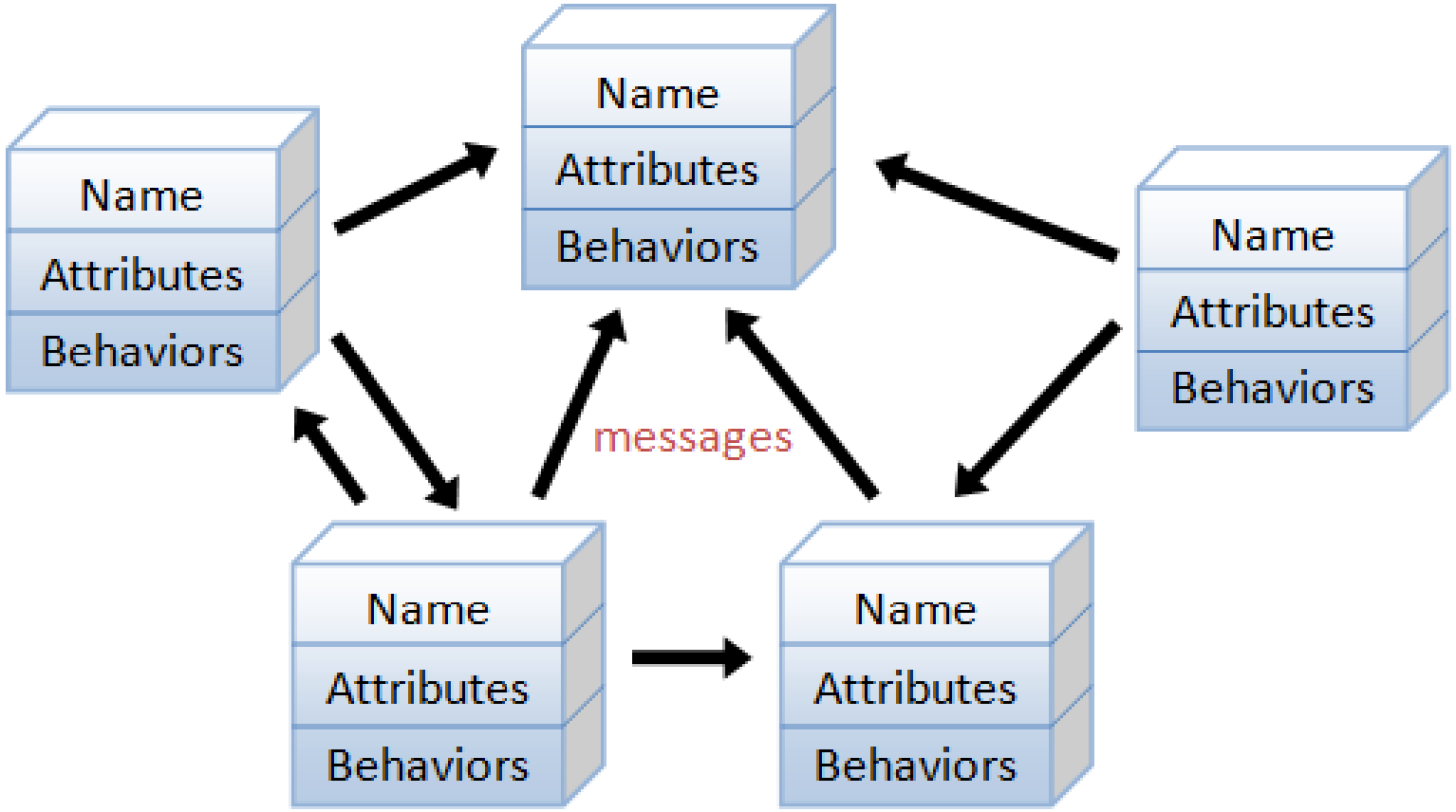
Le FUNZIONALITÀ di un oggetto sono solitamente rappresentate dai suoi metodi/funzioni che l'oggetto mette a disposizione degli altri oggetti come interfaccia di interazione

I metodi descrivono il COMPORTAMENTO dell'oggetto:

- ossia, il modo in cui un oggetto «risponde» a un messaggio ricevuto da un altro oggetto, modificando il proprio stato oppure interagendo con altri oggetti, per coordinare azioni o scambiare informazioni
- Interazione tramite messaggi:
  - L'invio di un messaggio è di solito codificato come chiamata di metodo
  - Risposta a un messaggio viene codificato come restituzione del risultato



L’esecuzione del programma

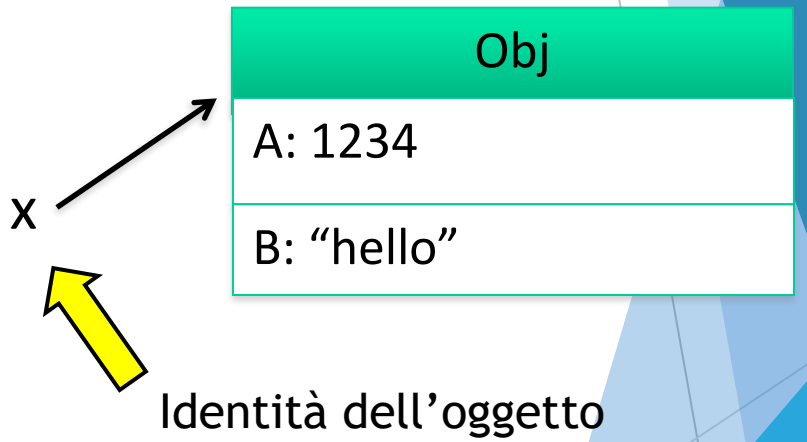


An object-oriented program consists of many well-encapsulated objects and interacting with each other by sending messages

Oggetti: caratteristiche

Oltre ad avere uno STATO e delle FUNZIONALITÀ, gli oggetti sono caratterizzati anche da:

- Identità (nome che individua l'oggetto)
- Ciclo di vita (creati, riferiti, disattivati)
- Locazione (di memoria)



Rispetto al paradigma imperativo:

- struttura dei programmi differente (ad es. insieme di classi)
- modello di esecuzione differente (ad es. memoria organizzata diversamente)

OOP: concetti

Oltre ad avere una nozione di oggetto, la programmazione object-oriented introduce una serie di concetti chiave

- Incapsulamento (già detto) e Astrazione (ragionare sul comportamento di un oggetto senza conoscerne la rappresentazione interna)
- Interfaccia (che cosa un oggetto mette a disposizione degli altri)
- Ereditarietà (come un oggetto acquisire le funzionalità di un altro oggetto, ad esempio estendendolo)
- Principio di sostituzione (quando un oggetto può essere usato al posto di un altro in maniera trasparente e controllata, senza alterare il funzionamento del programma)
- Polimorfismo (come un oggetto può elaborare altri oggetti indipendentemente anche di «tipi» diversi in modo uniforme, adattandosi dinamicamente in base al tipo dell'oggetto con cui stanno interagendo)

Questi sono concetti presenti in tutti i linguaggi object-oriented, anche se possono essere realizzati in modi diversi nei diversi linguaggi

Strutture linguistiche per l'OOP

- Dal punto di vista dei costrutti linguistici, i linguaggi di programmazione supportano i concetti dell'OOP seguendo due approcci principali
  - **Object-based** (JavaScript <2015, VisualBasic <2008, Self, Lua, …)
  - **Class-based** (Smalltalk, C++, Java, C#, Scala, …)

- JavaScript dal 2016 supporta entrambi gli approcci…
- Anche OCaml segue un approccio a cavallo tra le due filosofie

Approccio «object-based» all'OOP

- In un linguaggio **object-based**, gli oggetti vengono trattati nel linguaggio in maniera simile ai record
- I campi (detti anche membri/proprietà/variabili) rappresentano i dati interni dell'oggetto e possono essere associati a funzioni
- Una funzione in un oggetto (cioè, un metodo) può accedere ai campi dell'oggetto stesso tramite il riferimento this, che rappresenta l'istanza corrente
- Alcuni linguaggi, come JavaScript, consentono inoltre di modificare la struttura dell'oggetto dinamicamente (ad es. aggiungendo campi), offrendo così una maggiore flessibilità ma anche potenziali rischi per la robustezza del codice

Esempio JavaScript object-based

Si definisce un oggetto chiamato **mario** che rappresenta una persona con alcune proprietà e un metodo che incrementa di 1 l'età, usando **this**

```
let mario = {
  nome : "Mario",
  cognome : "Rossi",
  eta : 35,
  compleanno : function() {
    this.eta += 1;
  }
}

console.log(mario.nome); // Mario
console.log(mario.eta);  // 35
mario.compleanno();
console.log(mario.eta);  // 36
```

Esempio JavaScript object-based

In Javascript si possono **modificare dinamicamente** gli oggetti

```
// aggiungo dinamicamente un metodo
mario.nomeCompleto = function() {
  return this.nome + " " + this.cognome;
}

console.log(mario.nomeCompleto()); // Mario Rossi
```

Una volta aggiunto, il metodo `nomeCompleto` diventa parte dell'oggetto `mario` e può essere chiamato come qualsiasi altro metodo nativo dell'oggetto

Esempio JavaScript object-based

Per creare un oggetto è anche possibile definire una funzione costruttore, utilizzando la parola chiave **new**

```
function Persona(n, c, e) {
  this.nome = n;   this.cognome = c;   this.eta = e;
  this.compleanno = function() { this.eta++; }
  this.nomeCompleto = function() {
    return this.nome + " " + this.cognome;
  }
}
anna = new Persona("Anna","Rossi",33) ;
console.log(anna.nomeCompleto());    // A
anna.compleanno();
console.log(anna.eta);                // 34
```

Usando **new**, la funzione **Persona** crea istanze come **anna**, con proprietà e metodi
Si possono creare molteplici oggetti **Persona** con stato e comportamento indipendenti

Approccio «class-based» all'OOP

- In un linguaggio **class-based**, il concetto di classe è fondamentale e viene supportato da specifici costrutti linguistici.
- Una classe definisce la struttura e il comportamento di un insieme di oggetti di un determinato tipo, specificando
  - Le variabili (o proprietà) che rappresentano lo stato degli oggetti
  - I metodi che definiscono le funzionalità e il comportamento degli oggetti
- Gli oggetti vengono creati successivamente come istanze di una certa classe, ciascuno con il proprio stato e con accesso ai metodi definiti dalla classe

Esempio JavaScript class-based

```
class Persona {
  constructor(n,c,e) {
    this.nome=n; this.cognome=c; this.eta=e;
  }
  compleanno() { this.eta++; }
  nomeCompleto() {
    return this.nome + " " + this.cognome;
  }
}
rosa = new Persona("Rosa","Bianchi",25) ;
console.log(rosa.nomeCompleto()); // Rosa Bianchi
rosa.compleanno();
console.log(rosa.eta);            // 26
```

```
function Persona(n, c, e) {
  this.nome = n; this   .cognome = c;   this.eta = e;
  this.compleanno = function() { this.eta++; }
  this.nomeCompleto = function() {
    return this.nome + " " + this.cognome;
  }
}
anna = new Persona("Anna","Rossi",33) ;;
console.log(anna.nomeCompleto());    // Anna Rossi
anna.compleanno();
console.log(anna.eta);               // 34
```

La classe **Persona** viene definita con il costruttore **constructor(n,c,e)** che viene chiamato automaticamente quando si crea una nuova istanza della classe con **new**

object-based VS class-based

L’approccio **object-based**:

- consente al programmatore di lavorare con gli oggetti in modo flessibile
  - Gli oggetti possono essere creati direttamente, senza dover prima scrivere il codice della classe
  - È possibile creare facilmente tante varianti di uno stesso tipo di oggetto (ad es. con proprietà o metodi differenti), senza dover scrivere tante classi diverse
- permette di modificare la struttura dell'oggetto a tempo di esecuzione, aggiungendo o rimuovendo metodi e proprietà secondo necessità
- rende difficile predire con precisione quello che sarà il tipo di un oggetto, data l’assenza di classi predefinite
  - La struttura dell’oggetto può cambiare a tempo di esecuzione
  - Ostacola i controlli di tipo statici…

object-based VS class-based

L'approccio **class-based**:

- richiede al programmatore una maggiore disciplina:
  - Deve definire e implementare le classi prima di poter creare oggetti. Ogni oggetto è quindi un'istanza di una classe predefinita, e le relazioni tra classi sono chiare e strutturate
- consente di fare controlli di tipo statici sugli oggetti
  - il tipo di un oggetto è determinato: sarà legato alla classe da cui è stato istanziato
  - poiché il tipo di un oggetto è legato alla sua classe, questo modello si chiama nominal typing: l'appartenenza di un oggetto a un tipo dipende infatti dal nome della classe da cui deriva

È una scelta di design del linguaggio di programmazione:

- dipende da che tipo di utilizzo ci si aspetta sia fatto del linguaggio di programmazione in questione

Inheritance (Ereditarietà) e subtyping

La scelta tra object-based e class-based ha un impatto significativo sui meccanismi di **ereditarietà** e **(sotto)tipatura** del linguaggio:

- prototype-based inheritance vs class-based inheritance
- structural (sub)typing vs nominal (sub)typing

Inheritance (Ereditarietà)

L'ereditarietà è una funzionalità realizzata tramite opportuni costrutti linguistici che consente di definire una classe (o, più in generale, una tipologia di oggetti) sulla base di un'altra esistente

- I linguaggi **object-based**, per ogni oggetto mantengono una lista di prototipi, che sono tutti gli oggetti da cui esso eredita proprietà e metodi
- La catena di prototipi consente di creare strutture dinamiche, dove un oggetto può estendere o modificare le funzionalità ereditate senza dover definire una nuova classe

Inheritance (Ereditarietà) - prototipi

```
// Costruttore di Studente che eredita da Persona
function Studente(m, n, c, e) {
  this.matricola = m;
  this.__proto__ = new Persona (n,c,e);  // prototipo
}


let luigi = new Studente ("1231","Luigi","Verdi",22);
console.log(luigi.matricola);          // 1231
console.log(luigi.nomeCompleto());     // Luigi Verdi
```

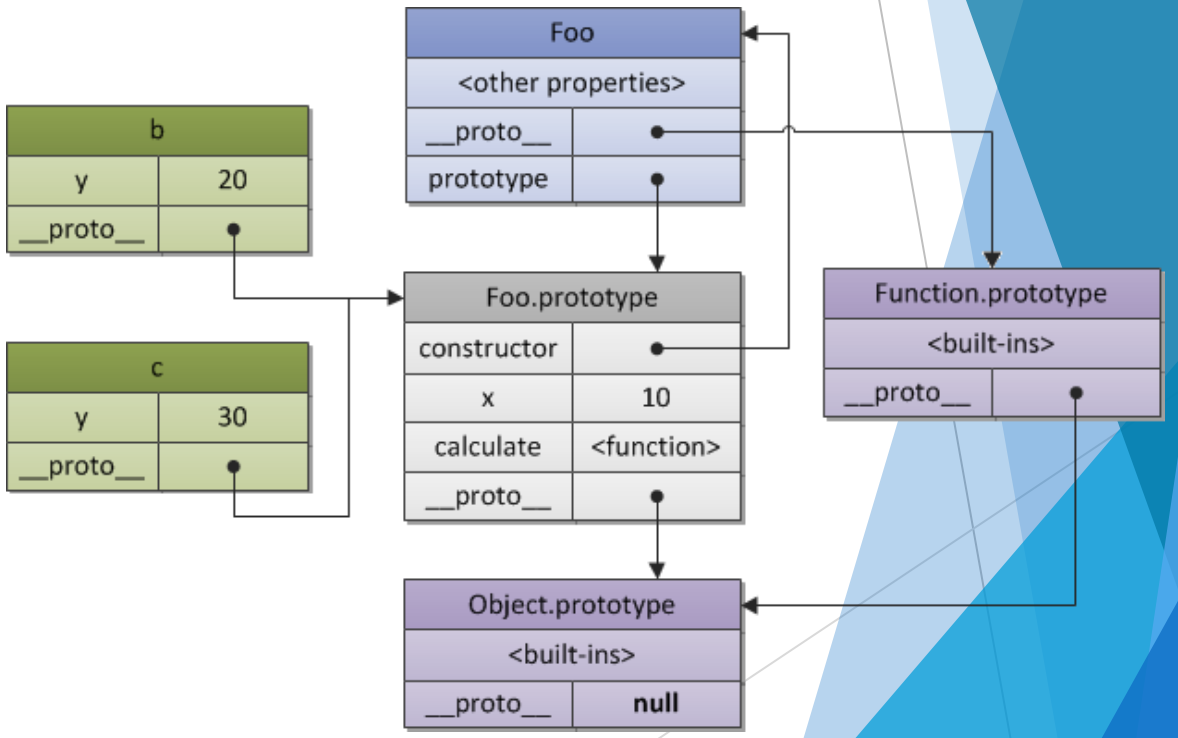
Imposta il prototipo di **Studente** come un'istanza di **Persona**, creata con **new Persona (n,c,e)**

Tutti i metodi e le proprietà di **Persona** saranno accessibili tramite **Studente** attraverso la catena dei prototipi

Inheritance (Ereditarietà) - prototipi

La gestione dei prototipi nei programmi diventa rapidamente piuttosto complicata…

- ogni oggetto può avere un prototipo, che a sua volta può avere il proprio prototipo, creando una catena di ereditarietà. Se la catena diventa lunga, risalire per capire da dove proviene una proprietà o un metodo può essere complicato



Inheritance (Ereditarietà)

- I linguaggi **class-based**, consentono di definire una classe come estensione di un'altra
- La nuova classe eredita automaticamente tutti i membri (proprietà e metodi) della precedente, con la possibilità di:
  - aggiungerne altri
  - o ridefinirne alcuni, tramite l'overriding: questo meccanismo permette alla nuova classe di sostituire il comportamento di un metodo ereditato con una nuova implementazione, mantenendo il nome e la firma del metodo

Inheritance (Ereditarietà) - estensione

```
// Classe Studente che estende la classe Persona
class Studente extends Persona {
  constructor(m,n,c,e) {
    super (n,c,e);            // Studente acquisisce i campi di Persona
    this.matricola = m;       // NO lista prototipi
  }
}

let giada = new Studente("7212","Giada","Neri",21);
console.log(giada.matricola);          // 7212
console.log(giada.nomeCompleto());     // Giada Neri
```

La chiamata a `super (n,c,e)` invoca il costruttore della classe **Persona** e inizializza le proprietà dell'oggetto **Studente**, in base a quelle definite in Persona

Con `this.matricola` si aggiunge una nuova proprietà `matricola` specifica della classe **Studente**, inizializzata con il valore **m** passato al costruttore

Inheritance e subtyping

I meccanismi di inheritance (e in generale il fatto di avere oggetti che sono l'uno una «estensione» dell'altro) inducono nozioni di sottotipo tra oggetti

Idealmente, un oggetto B, che è estensione di un altro oggetto A, dovrebbe poter essere usato dovunque si possa usare A

- Un oggetto che descrive uno studente dovrebbe poter essere usato ovunque sia richiesto un oggetto che descrive genericamente una persona
- Il tipo «studente» dovrebbe essere un sottotipo di «persona», così che tutti i metodi e le proprietà disponibili per «persona» siano accessibili anche per «studente»

Structural subtyping

I linguaggi **object-based** solitamente usano una nozione di **subtyping strutturale**:

- un oggetto B è sottotipo di un oggetto A se contiene almeno tutti i membri «pubblici» (variabili e metodi utilizzabili dall'esterno… vedremo) che sono presenti anche in A
- un oggetto Studente è considerato un sottotipo di Persona se contiene tutti i campi e metodi presenti in Persona, oltre a eventuali estensioni specifiche di Studente

Nominal subtyping

I linguaggi **class-based** solitamente usano una nozione di **subtyping nominale**:

- il tipo di un oggetto è determinato dalla classe da cui è stato istanziato
- il nome della classe diventa il nome del tipo
- un tipo-classe B è sottotipo di un tipo-classe A se la classe B è stata definita esplicitamente (sintatticamente) come estensione della classe A (ad esempio, con `class B extends A`)
  - vale la proprietà transitiva per la relazione di sotto-tipo: se C estende B e B estende A, allora C è sottotipo di B e anche di A.
- uno studente è sottotipo di una persona, perché «Studente extends Persona»

Structural VS Nominal Subtyping

Structural Subtyping

- Maggiore flessibilità: non è necessario dichiarare esplicitamente chi estende chi (la relazione di sottotipo è basata sulla somiglianza tra le strutture degli oggetti)
- Supporto al polimorfismo:(la relazione di sottotipo è più "debole" , quindi ci sono più oggetti l'uno sottotipo dell'altro). . Due oggetti con strutture compatibili possono essere considerati l'uno sottotipo dell'altro

Nominal Subtyping

- Maggior rigore: mette in relazione di sottotipo **solo** classi che il programmatore ha esplicitamente dichiarato essere in questa relazione con extends (vincoli dichiarativi)
- Maggiore semplicità nella verifica per l'interprete: (deve solo controllare se una classe è nominata nella lista degli antenati dell'altra… senza confrontare il contenuto delle due classi)

Scelte diverse…

- **JavaScript** adotta lo structural subtyping, avendo una radice object-based
- **Java** è un linguaggio class-based e adotta il nominal subtyping
- **OCaml** è un linguaggio in cui gli aspetti di erediarietà sono trattati con costrutti linguistici class-based (extends), **ma** adotta lo structural subtyping
- Vedremo in dettaglio l'approccio di **Java** (che è quello più puramente **class-based**), ma
- diamo prima un'occhiata agli **oggetti di OCaml** (per vedere bene lo **structural subtyping** con controlli statici dei tipi)

Uno sguardo (non completo) all'OOP in OCaml

Vedere anche i capitoli relativi sul libro «Real World OCaml»

Oggetti, classi e tipi oggetto

- Un oggetto in OCaml è un valore, costituito da campi e metodi che rappresentano rispettivamente lo stato e il comportamento dell'oggetto
- Sebbene esistano costrutti linguistici per la definizione di classi, gli oggetti possono essere creati direttamente, senza prima specificare una classe (come in JavaScript)
- Il tipo di un oggetto è determinato esclusivamente dai metodi che esso contiene ed espone (i campi non influiscono sul tipo)

Esempio di oggetto (stack)

```
(* oggetto che realizza uno stack *)
let s = object

    (* campo mutabile che contiene la rappresentazione dello stack *)
    val mutable v = [0; 2]   (* Assumiamo per ora inizializzato non vuoto *)

    (* metodo pop *)
    method pop =
        match v with
        | hd :: tl ->
            v <- tl;
            Some hd
        | [] -> None

    (* metodo push *)
    method push hd =
        v <- hd :: v
end ;;
```

v: stato interno (modificabile) dello stack

Esempio di oggetto (stack)

```
(* oggetto che realizza uno stack *)
let s = object

    (* campo mutabile che contiene la rappresentazione de
    val mutable v = [0; 2]   (* Assumiamo per ora iniziali

    (* metodo pop *)
    method pop =
        match v with
        | hd :: tl ->
            v <- tl;
            Some hd
        | [] -> None

    (* metodo push *)
    method push hd =
        v <- hd :: v
end ;;
```

NOTA SINTATTICA 1:
Nei metodi senza parametri non è necessario aggiungere ()

NOTA SINTATTICA 2:
I campi dell'oggetto sono visibili nei metodi (non serve this)

TIPO INFERITO (object-type, solo metodi):

val s : < pop : int option; push : int -> unit > = <obj>

I nomi dei metodi appaiono in ordine alfabetico per rendere efficienti i confronti

REPL: uso dell'oggetto s

Dopo aver definito l'oggetto s, è possibile interagire con esso chiamando i metodi push e pop:

```
s#pop ;;
- : int option = Some 0
s#pop ;;
- : int option = Some 2
s#pop ;;
- : int option = None
s#push 9 ;;
- : unit = ()
s#pop ;;
- : int option = Some 9
```

NOTA SINTATTICA:
Invocazione di metodo si fa con #-notation invece che con dot-notation (no overloading)

Piccola digressione: type weakening

Non è relativo agli oggetti, ma al type inference di variabili mutabili

Piccola digressione: type weakening

Domanda:

che succede se nell'oggetto **s** inizializziamo **v** come lista vuota?

Che tipo viene inferito per l'oggetto?

```
let s = object

   val mutable v = []   (* lista vuota! *)

   method pop = ...
   method push hd = ...
end ;;
```

Piccola digressione: type weakening

Domanda:

che succede se nell'oggetto **s** inizializziamo **v** come lista vuota?

Che tipo viene inferito per l'oggetto?

```
let s = object

    val mutable v = []   (* lista vuota!! *)

    method pop = ...
    met
end ;;
```

TIPO INFERITO

val s : < pop : '_weak option; push : '_weak -> unit > = <obj>

Piccola digressione: type weakening

TIPO INFERITO

val s : < pop : '_weak option; push : '_weak -> unit > = <obj>

Il tipo inferito contiene variabili di tipo, ma non è veramente polimorfo…

Lo è solo temporaneamente

- Benché sia mutabile, la variabile **v** non potrà avere tipi diversi in momenti diversi dell'esecuzione

```
val mutable v = []
```

- L'oggetto **s** dovrebbe avere tipo

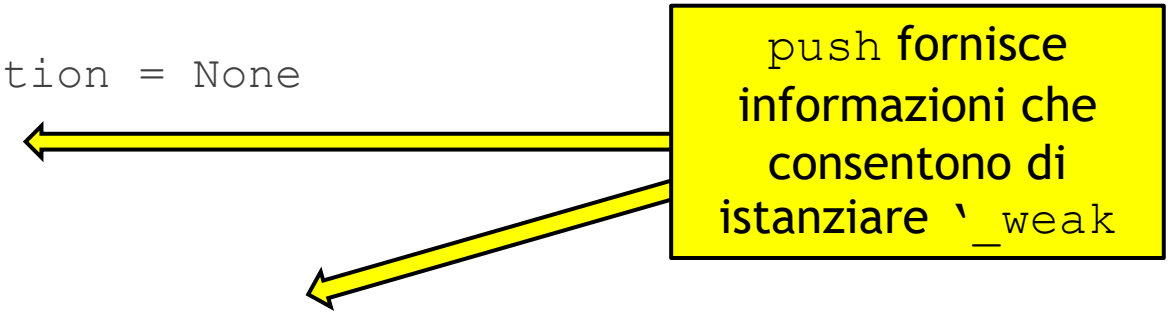
  < pop : t option; push : t -> unit > per un qualche tipo concreto t

- t non è però noto al momento della dichiarazione della variabile mutabile, quindi il type checker indebolisce temporaneamente il tipo inferito includendo delle variabili di tipo
- Appena possible (al primo utilizzo) il tipo di s sarà ricalcolato andando a istanziare definitivamente la variabile provvisoria con un tipo concreto

REPL: uso dell'oggetto con type weakening

```
s;;
val s : < pop : '_weak option; push : '_weak -> unit > = <obj>
s#pop ;;
- : '_weak option = None
s#push 5 ;;
- : unit = ()
s ;;
val s : < pop : int option; push : int -> unit > = <obj>
s#pop ;;
- : int option = Some 5
s#push "ciao" ;;
6 | s#push "ciao" ;;
            ^^^^^^

Error: This expression has type string but an expression was expected
of type
         int
```



Fine digressione

Riprendiamo a parlare di oggetti

Costruzione di oggetti tramite funzioni

il valore di `init` dovrebbe essere una lista, perché lo stack è implementato come una lista

- Gli oggetti possono essere costruiti tramite funzioni

```
(* funzione "stack" che costruisce oggetti inizializzati con init *)
let stack init = object
    val mutable v = init (* valore iniziale *)

    method pop =
        match v with
        | hd :: tl ->
            v <- tl;
            Some hd
        | [] -> None

    method push hd =
        v <- hd :: v
end ;;
```

TIPO INFERITO

```
val stack : 'a list ->
                < pop : 'a option; push : 'a -> unit >
                                                = <fun>
```

La funzione stack è (veramente) polimorfa!

REPL: uso della funzione stack

```
let s = stack [3; 2; 1] ;; (* Crea uno stack inizializzato con [3; 2; 1] *)
val s : < pop : int option; push : int -> unit > = <obj>
s#pop ;;                    (* Rimuove e restituisce 3 *)
- : int option = Some 3

let s = stack [] ;;  (* con [] ancora type weakening... *)
val s : < pop : '_weak option; push : '_weak -> unit > = <obj>

(* ... ma basta una type annotation del parametro attuale per forzare
l'instanziazione al tipo che preferiamo *)
let s = stack ([]: int list) ;;
val s : < pop : int option; push : int -> unit > = <obj>
```

Polimorfismo di oggetti

Quando si definisce una funzione che prende un oggetto come parametro, il tipo dell'oggetto viene inferito dai metodi che la funzione chiama su di esso

- Indipendentemente dal fatto che l'oggetto sia già stato definito o meno!

```
let area sq = sq#width * sq#width ;;
val area : < width : int; .. > -> int = <fun>

let minimize sq = sq#resize 1 ;;
val minimize : < resize : int -> 'a; .. > -> 'a = <fun>

let limit sq = if (area sq) > 100 then minimize sq ;;
val limit : < resize : int -> unit; width : int; .. > -> unit = <fun>
```

Polimorfismo di oggetti

Quando si definisce una funzione che prende un oggetto come parametro, il tipo dell'oggetto viene inferito dai metodi che la funzione chiama su di esso

- Indipendentemente dal fatto che l'oggetto sia già stato definito o meno!

```
let area sq = sq#width * sq#width ;;
val area : < width : int; .. > -> int = <fun>

let minimize sq = sq#resize 1 ;;
val minimize : < resize : int -> 'a; .. > -> 'a =

let limit sq = if (area sq) > 100 then minimize s
val limit : < resize : int -> unit; width : int; .. > -> unit = <fun>
```

Questa notazione indica che l'oggetto atteso, da passare ad `area` come parametro, deve contenere almeno il metodo `width`, ed eventualmente anche altro (espresso tramite i puntini `..`)

Polimorfismo di oggetti

Una funzione può operare su oggetti di tipi diversi, purché questi soddisfino i requisiti di metodo

Quando si definisce una funzione che prende un oggetto come parametro, il tipo dell'oggetto viene inferito dai metodi che la funzione chiama su di esso

- Indipendentemente dal fatto che l'oggetto sia già stato definito o meno!

```
let area sq = sq#width * sq#width ;;
val area : < width : int; .. > -> int = <fun>

let minimize sq = sq#resize 1 ;;
val minimize : < resize : int -> 'a; .. > -> 'a =

let limit sq = if (area sq) > 100 then minimize s
val limit : < resize : int -> unit; width : int;
```

Flessibilità: la funzione non richiede che l'oggetto sia stato definito in anticipo con uno specifico tipo.
Basta che l'oggetto passato alla funzione contenga i metodi richiesti, senza vincoli aggiuntivi

Polimorfismo di oggetti

Il seguente oggetto «quadrato» può essere passato a tutte le funzioni viste, poiché soddisfa i requisiti di tipo richiesti da ciascuna di queste funzioni

```
let quadrato = object
  val w = ref 30
  method width = !w
  method color = "red"
  method resize n = w := n
end ;;
```

- `area : < width : int; .. > -> int`
- `minimize : < resize : int -> 'a; .. > -> 'a`
- `limit : < resize : int -> unit;`
  `width : int; .. > -> unit`

```
val quadrato : < color : string; resize : int -> unit; width : int > = <obj>
```

Polimorfismo di oggetti: structural subtyping

Si applica la regola di subsumption

$$\frac{\Gamma \vdash e : S \quad S <: T}{\Gamma \vdash e : T}$$

che consente di tipare un'espressione con un suo **supertipo**

Ad esempio, abbiamo:

```
< color : string;
  resize : int -> unit;
  width : int >
```

$<:$

```
< width : int >
```

```
area : < width : int; .. > -> int
```

Grazie a cui, nel nostro esempio, possiamo derivare

$$\Gamma \vdash quadrato : \texttt{< width : int >}$$

e concludere che l'oggetto `quadrato` può essere passato alla funzione `area`. Lo stesso vale per le funzioni `minimize` e `limit`

Polimorfismo di oggetti: structural subtyping

La notazione con i puntini

```
< width : int, .. >
```

usata dall'interprete OCaml nell'inferire il tipo del parametro formale di `area` enfatizza lo structural subtyping

- La funzione accetta un qualunque sottotipo di `< width : int >`, ossia qualunque oggetto che contenga almeno il metodo `width`
- Rappresenta una nuova forma di polimorfismo (sugli oggetti) che consente di trattare diversi oggetti con la stessa funzione, purché abbiano i metodi richiesti, indipendentemente dagli altri metodi presenti
- I puntini `..` possono essere considerati come una variabile di tipo, che si può istanziare con una lista di altri metodi potenzialmente da aggiungere a `width`

REPL: uso della dell’oggetto quadrato

```
area quadrato ;;
- : int = 900
limit quadrato ;;
- : unit = ()
area quadrato ;;
- : int = 1
```

Coercion di tipi oggetto

Al di là del passaggio dei parametri a una funzione, ci sono numerose altre situazioni in cui il subtyping degli oggetti si rende utile

Supponiamo di definire i seguenti tipi oggetto (tramite `type`):

```
type shape = < area : float >
type square = < area : float; width : int >
```

È chiaro che `square` sia un sottotipo di `shape` (contiene dei metodi in più), quindi è lecito pensare che ovunque si possa usare un oggetto di tipo `shape` si possa usare al suo posto un oggetto di tipo `square` (PRINCIPIO DI SOSTITUZIONE, ne riparleremo…)

Coercion di tipi oggetto

Facciamo una prova… definiamo funzioni costruttore per i due tipi:

```
(* costruttore di oggetti di tipo shape *)
let shape (a:float): shape = object
  method area = a
end ;;

(* costruttore di oggetti di tipo square *)
let square (w:int): square = object
  method area = (float_of_int) (w * w)
  method width = w
end ;;
```

Coercion di tipi oggetto

Proviamo ad aggiungere uno `square` a una lista di `shape`:

```
let lis1  = [shape 10.0; shape 20.0] ;;
val lis1 : shape list = [<obj>; <obj>]

let lis2  = square 5 :: lis1 ;;
1 | let lis2 = (square 5) :: lis1
                             ^^^^
Error: This expression has type shape list
       but an expression was expected of type square list
       Type shape = < area : float > is not compatible with type
         square = < area : float; width : int >
       The first object type has no method width
```

NON FUNZIONA...

Coercion di tipi oggetto: operatore :>

Serve una type coerction (conversione di tipo) esplicita, tramite l'operatore :>

```
let lis2  = ( square 5 :> shape ) :: lis1 ;;
val l2 : shape list = [<obj>; <obj>; <obj>]
```

ORA FUNZIONA!

La type coercion `e :> t` forza il type checker a trattare l'espressione `e` come se fosse di tipo `t`

- `t` deve essere un tipo più generale (un **supertipo**, con metodi in meno) del tipo originale di `e`

Ad esempio:

```
let (x:shape) = ((square 2) :> shape) ;;      (* OK *)
let (y:square) = ((shape 4.0) :> square) ;; (* ERRORE!! *)
```

Polimorfismo di oggetti VS principio di sostituzione

Questo esempio mostra che i due concetti di

- polimorfismo sugli oggetti

  (ad es. una funzione che prende oggetti con almeno i metodi richiesti)

- principio di sostituzione

  (ad es. un oggetto di un tipo più specifico si può usare ovunque serva un oggetto di un tipo più generale)

sebbene tra loro collegati, vengono trattati in OCaml in due modi diversi:

il primo è supportato direttamente grazie al **subtyping strutturale**, mentre per il secondo è richiesta infatti la type coercion esplicita

- Questo sempre perché il type checker di OCaml, come già visto con i tipi primitivi, non effettua conversioni di tipo implicite

Type coerction di oggetti: OCaml VS Java

Narrowing

In Java vedremo che sarà possibile effettuare delle **coercizioni di tipo** da un supertipo a un sottotipo (ad es. trasformare uno `shape` in uno `square`)

- Questo sarà possibile grazie ai **controlli dinamici di tipo** eseguiti dall'interprete della JVM a run time, che risultano più semplici grazie all'uso del nominal subtyping

```
// Frammento di codice Java
Shape s = getShape();
if (s instanceOf Square) {
    int w = ((Square)s).width();
    System.out.println("Quadrato di lato " + w);
}
else System.out.println("Non è un quadrato");
```

Classi
(e costrutti linguistici per l'ereditarietà)

- Abbiamo visto che OCaml consente di lavorare direttamente con gli oggetti (in stile object-based)
- Tuttavia, i meccanismi di forza della programmazione OO derivano dall'ereditarietà
- Abbiamo visto in JavaScript che realizzare meccanismi di ereditarietà lavorando direttamente con gli oggetti richiede di usare tecniche tipo i prototipi, il cui funzionamento è complicato…
- Per questo OCaml introduce **anche** dei costrutti di classe

Classi

Intuitivamente, una classe è la «ricetta» che descrive come creare oggetti di un certo tipo

- una classe si definisce con `class` e si istanzia con `new`

```
class istack = object  (* classe per stack di interi *)
  val mutable v = [0; 2]  (* inizializzato non vuoto *)
  method pop =
    match v with
    | hd :: tl ->
        v <- tl;
        Some hd
    | [] -> None
  method push hd =
    v <- hd :: v
end ;;
```

```
let s = new istack ;;
val s : istack = <obj>
s#pop ;;
- : int option = Some 0
```

`istack`: alias per il tipo

Classi parametriche e polimorfe

Una classe può prevedere parametri di

- costruzione (ad es. `init`) da passare al momento dell'istanziazione
- tipo (ad es. `'a`) che la rendono polimorfa

```
class ['a] stack init = object  (* classe polimorfa per stack *)
  val mutable v : 'a list = init   (* init è parametro costruttore *)
  method pop =
    match v with
    | hd :: tl ->
        v <- tl;
        Some hd
    | [] -> None
  method push hd =
    v <- hd :: v
end ;;
```

`['a] stack` indica che **stack** è una classe polimorfica che funziona con liste di tipo 'a, rendendola generica

```
let s = new stack ["pippo"] ;;
val s : string stack = <obj>
s#pop ;;
- : string option = Some "pippo"
```

Classi e tipi oggetto

La definizione di una classe introduce anche un tipo con lo stesso nome

- Si tratta però solo di un alias del tipo-oggetto che si otterrebbe costruendo gli oggetti direttamente

```
let s = new stack ["pippo"] ;;
val s : string stack = <obj>
(* string stack è un alias per
   < pop : string option; push : string -> unit >  *)
```

Inheritance (Ereditarietà) – ripetiamo…

L'ereditarietà è una funzionalità realizzata tramite opportuni costrutti linguistici che consente di definire una classe (o, più in generale, una tipologia di oggetti) sulla base di un'altra esistente

- I linguaggi class-based, consentono di definire una classe come estensione di un'altra
- La nuova classe:
  - eredita tutti i membri (valori e metodi) della precedente,
  - con la possibilità di aggiungerne altri (o ridefinirne alcuni, overriding)

Inheritance (Ereditarietà)

Esempio:

```
class sstack init = object  (* classe per stack di stringhe *)
   inherit [string] stack init (* eredita da stack *)

   method concat =                  (* aggiunge un nuovo metodo *)
      List.fold_left (^) v
end ;;
```

```
let b = new sstack [" ";"world!"] ;;
val b : sstack = <obj>
b#push "Hello" ;;
-  : unit = ()
b#concat ;;
- : string = "Hello world!"
```

Overriding

Esempio:

```
(* classe per stack di int che raddoppia i valori inseriti *)
class double_stack init = object
   (* super è l'oggetto da estendere in fase di istanziazione *)
    inherit [int] stack init as super
```

Creo l'alias **super**

```
    method push hd =                    (* ridefinisce un metodo *)
        super#push ( hd * 2 )
end ;;
```

```
let ds = new double_stack [] ;;
val ds : double_stack = <obj>
ds#push 5 ;;
- : unit = ()
ds#pop ;;
- : int option = Some 10
```

OOP in OCaml: recap

OCaml combina costrutti linguistici tipicamente object-based con costrutti tipicamente class-based

- I costrutti object-based favoriscono la definizione di object-types e supportano lo structural subtyping
- La non modificabilità della struttura degli oggetti (a differenza di JavaScript) consente tuttavia di effettuare controlli di tipo a tempo di compilazione
- I costrutti class-based rendono naturale la definizione di meccanismi di ereditarietà (e altro…), garantendo una gerarchia chiara tra le classi

OOP in OCaml: altri aspetti

OCaml prevede anche altri costrutti di OOP, che consentono di trattare aspetti importanti quali:

- Interfacce
- Classi parzialmente definite (classi astratte)
- Iteratori
- …

Non vedremo tutti questi aspetti, ma li tratteremo in Java