

# PROGRAMMAZIONE ORIENTATA AGLI OGGETTI

Object-Oriented Programming (OOP)

# Modularità dei programmi

I linguaggi di programmazione supportano in vari modi la possibilità di **modularizzare** i programmi. Ad esempio:

- ▶ Sotto il **profilo linguistico**, con **l'astrazione procedurale** (possibilità di scomporre il problema in sottoproblemi da risolvere con specifiche procedure/funzioni)

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

# Modularità dei programmi

Ad esempio:

- Sotto il **profilo dei tipi di dato**, con i **tipi di dato astratti**

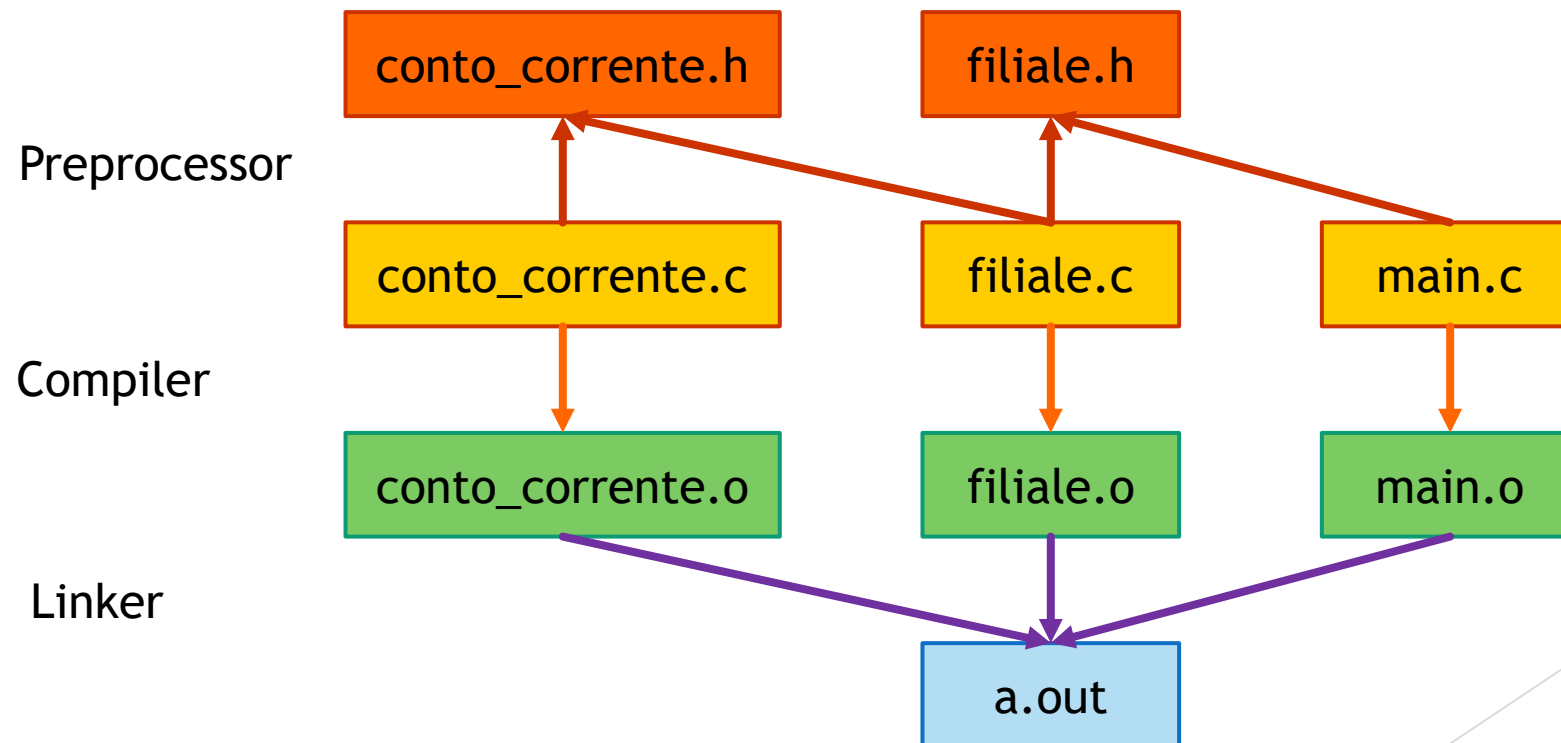
```
module type BOOL = sig
  type t
  val yes: t
  val no: t
  val choose: t -> 'a -> 'a -> 'a
```

```
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
```

# Modularità dei programmi

Ad esempio:

- Sotto il **profilo delle tecniche di compilazione** ed esecuzione, con la **compilazione separata** e il linking



# Livelli di astrazione

La possibilità di modularizzare i programmi consente di progettare e sviluppare un programma per **livelli di astrazione**

Ad esempio:

- ▶ La **libreria standard** consente di scrivere un programma che opera su stringhe astraendo da (i.e., ignorando) come le operazioni su stringhe siano implementate
- ▶ Per implementare un programma di **gestione di banche**:
  1. Si implementano 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. Si passa a implementare i moduli di gestione della **rete di filiali** usando il modulo della singola filiale, ma astraendo dalla sua implementazione

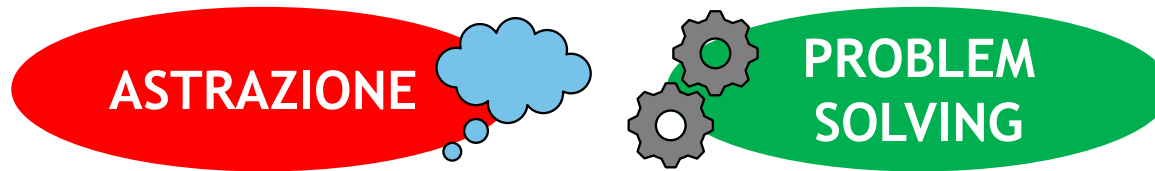
# Livelli di astrazione

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

- ▶ Linguaggi di programmazione  
source code -> bytecode -> assembler
- ▶ Sistemi operativi  
applicazione -> sistema operativo -> hardware
- ▶ Protocolli di comunicazione su reti  
http -> tcp -> ip -> Ethernet

# Astrazione + problem solving

Le capacità di



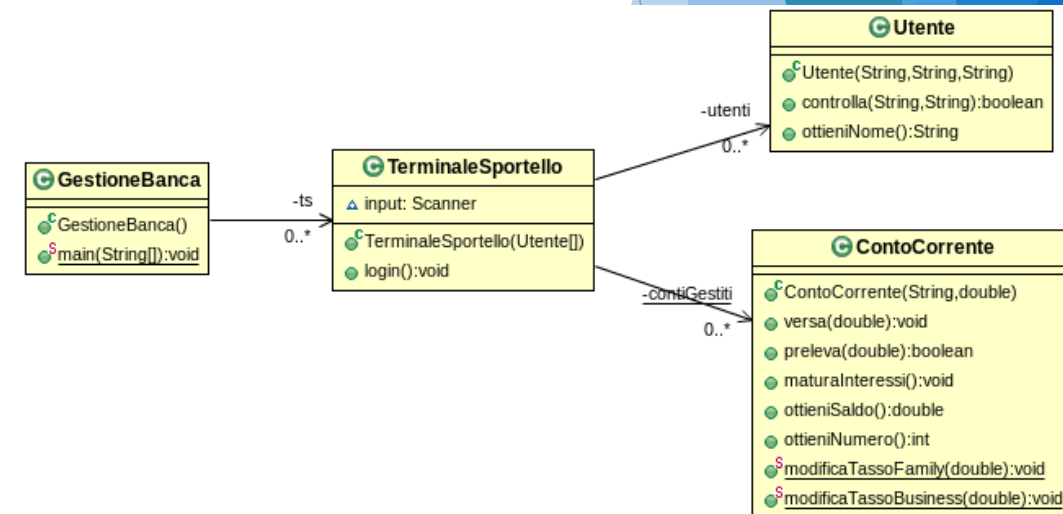
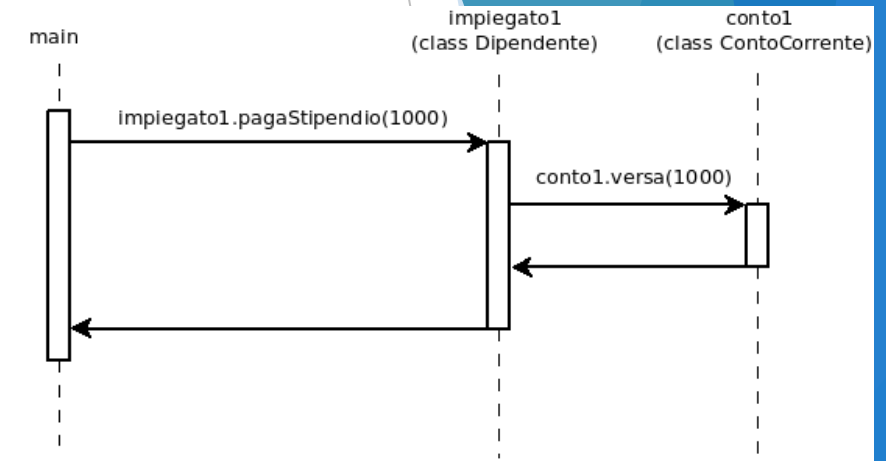
caratterizzano un **buon informatico**

**CONSENTONO DI TROVARE  
SOLUZIONI EFFICIENTI  
A PROBLEMI COMPLESSI**

# 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
- ▶ Ognuno sviluppa la 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 usando diagrammi standard (UML)

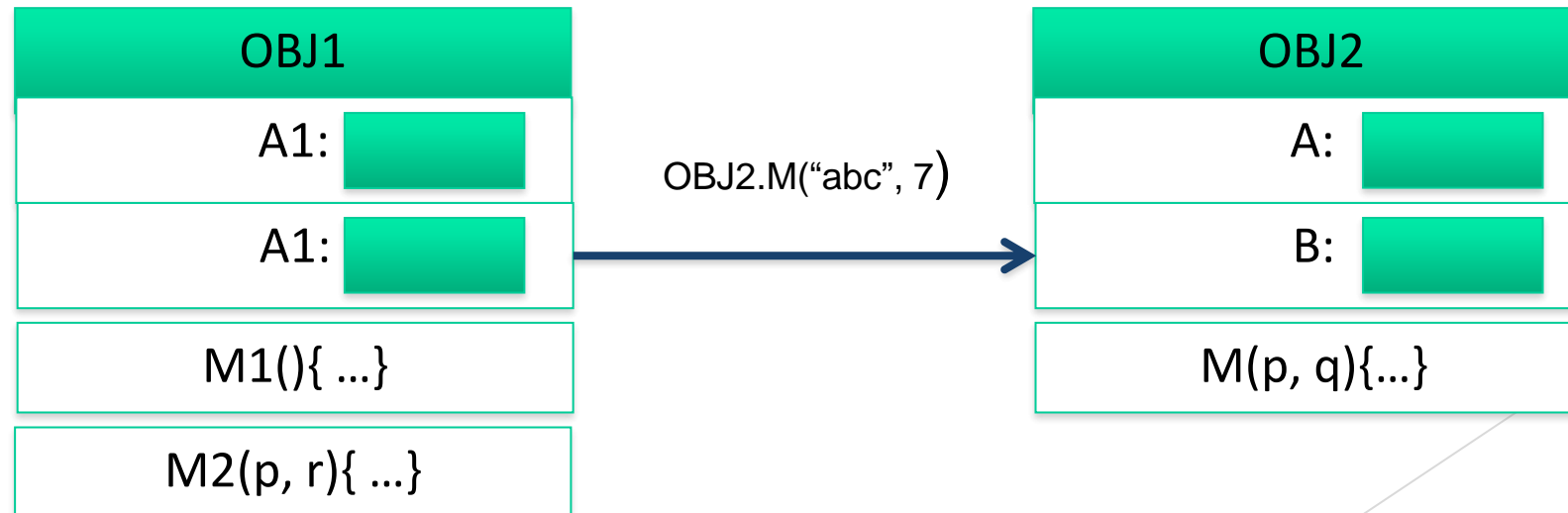




# IL PARADIGMA OBJECT-ORIENTED

# Il paradigma a oggetti

- ▶ Sistema software = insieme di **oggetti cooperanti**
- ▶ Gli oggetti sono caratterizzati da:
  - ▶ Uno **STATO**
  - ▶ Un insieme di **FUNZIONALITA'**
- ▶ Gli oggetti cooperano scambiandosi **messaggi**



# Lo STATO di un oggetto

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

- Proprietà di **INCAPSULAMENTO**

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

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

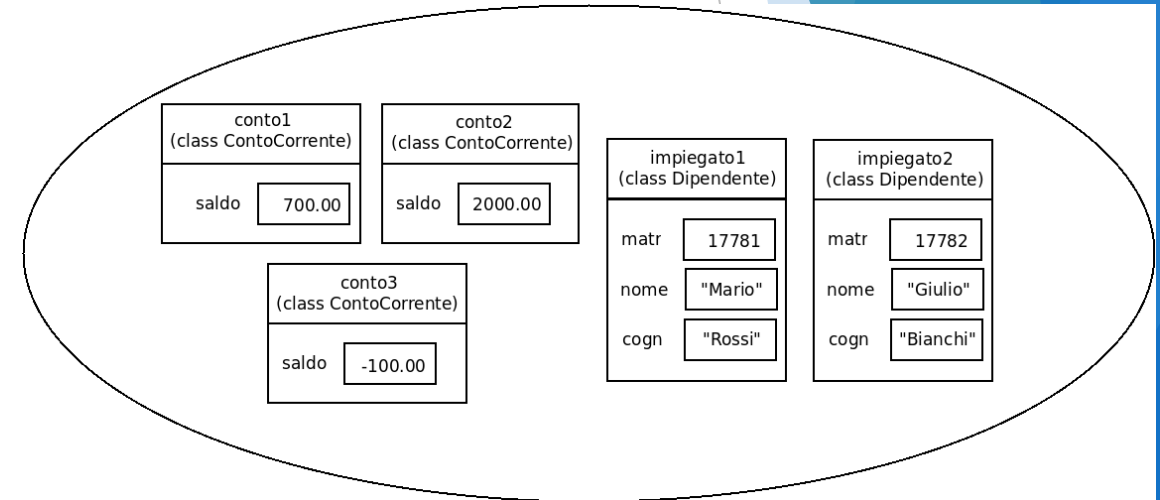
- Anzi, A non dovrebbe nemmeno aver bisogno di sapere come lo stato di B sia rappresentato (cioè, che variabili usa)...

| Persona1        |         |
|-----------------|---------|
| Nome:           | "Mario" |
| Età:            | 35      |
| getNome(){ ...} |         |
| incrEtà(){ ...} |         |

# Lo stato del programma

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

- 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 FUNZIONALITA' di un oggetto

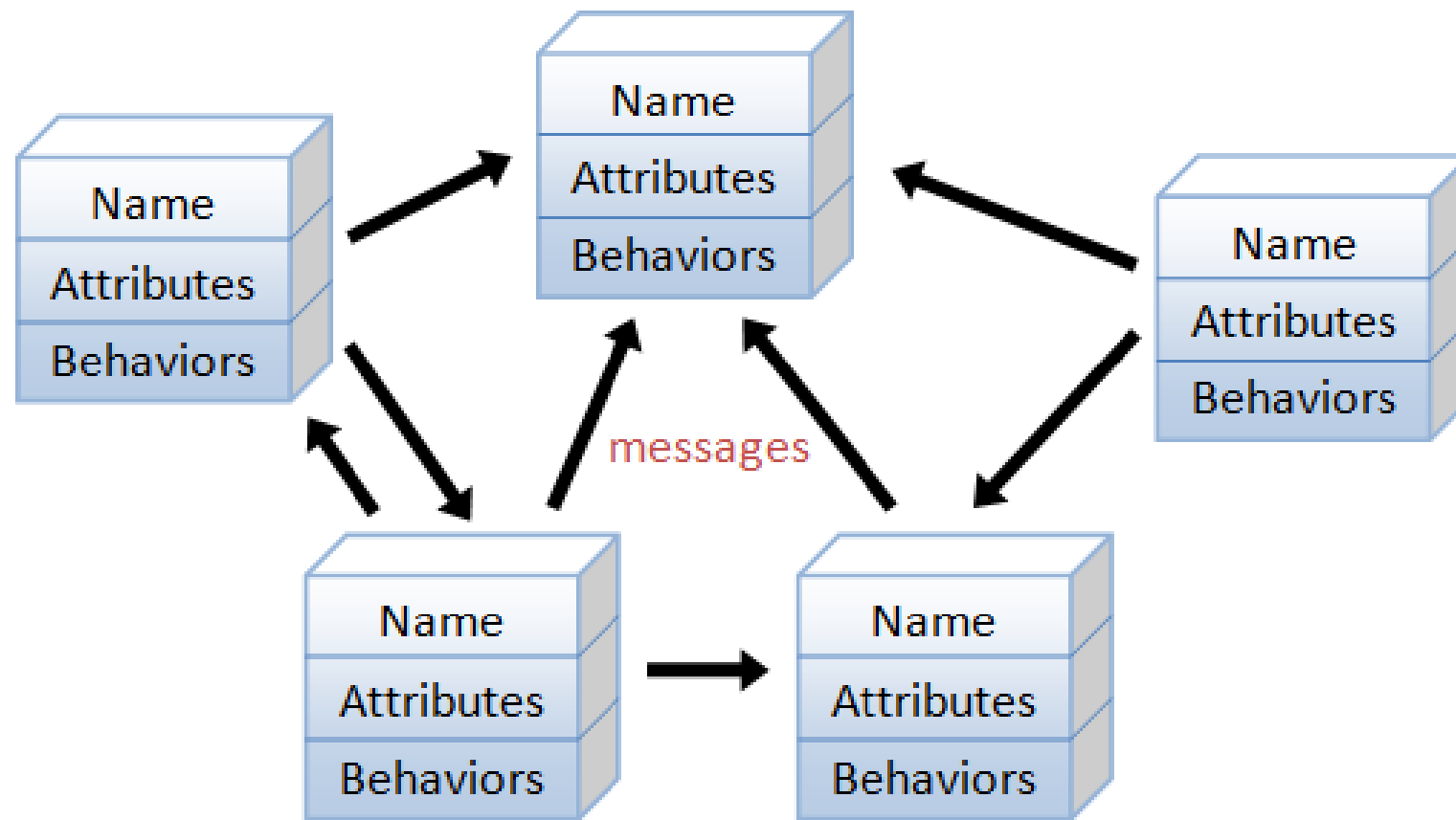
Le **FUNZIONALITA'** di un oggetto sono solitamente **rappresentate** da un gruppo di **metodi/funzioni** che l'oggetto **mette a disposizione degli altri oggetti**

I metodi descrivono il **COMPORTAMENTO** dell'oggetto:

- ▶ Ossia, come un oggetto «risponde» ad un messaggio ricevuto da un altro oggetto, anche modificando il proprio stato o interagendo con altri oggetti
- ▶ Solitamente:
  - ▶ **Invio di un messaggio** codificato come **chiamata di metodo**
  - ▶ **Risposta ad un messaggio** codificato come **restituzione del risultato**

| Persona1        |         |
|-----------------|---------|
| Nome:           | "Mario" |
| Età:            | 35      |
| getNome(){ ...} |         |
| incrEtà(){ ...} |         |

# L'esecuzione del programma



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

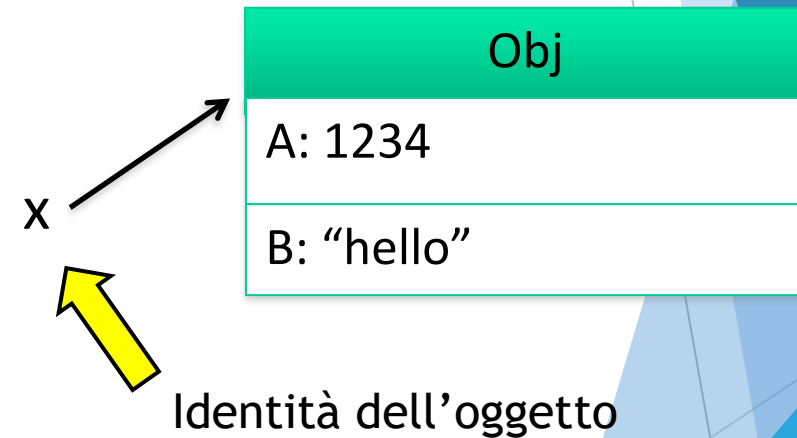
# Oggetti: caratteristiche

Oltre ad avere uno **STATO** e delle **FUNZIONALITA'**, 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:

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



# OOP: concetti

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

- ▶ **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 può fare proprie 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)
- ▶ **Polimorfismo** (come un oggetto può **processare** altri oggetti indipendentemente anche di «**tipi**» **diversi**)

Questi sono concetti presenti in tutti i linguaggi object-oriented, ma che 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, 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

- ▶ Gli oggetti vengono trattati nel linguaggio in maniera **simile ai record**
- ▶ I **campi** (detti anche **membri/proprietà/variabili**) possono essere associati a funzioni
- ▶ Una funzione in un oggetto (cioè, un **metodo**) può accedere ai campi dell'oggetto stesso tramite il riferimento **this**.
- ▶ **JavaScript**, ad esempio, consente inoltre di **modificare** la struttura dell'oggetto dinamicamente (es. aggiungendo campi)

# Esempio JavaScript object-based

```
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

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

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

# Esempio JavaScript object-based

Per creare un oggetto è anche possibile definire una **funzione costruttore**, da richiamare con **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());    // Anna Rossi  
anna.compleanno();  
console.log(anna.eta);                // 34
```

# Approccio «class-based» all'OOP

- ▶ Un linguaggio «class-based» prevede un concetto di «**classe**» a cui corrispondono determinati costrutti linguistici
- ▶ Una **classe** **definisce** il contenuto (variabili e metodi) degli oggetti di un certo **tipo**
- ▶ Gli **oggetti** vengono creati successivamente come **istanze** di una certa 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
```

# object-based VS class-based

L'approccio **object-based**:

- ▶ Consente al programmatore di lavorare con gli oggetti in modo **flessibile**:
  - ▶ Non è necessario scrivere il codice della classe prima di creare un oggetto
  - ▶ Si può creare tante varianti di un oggetto (ad es. con metodi diversi) senza bisogno di scrivere tante classi diverse
- ▶ Rende **difficile predire** con precisione quello che sarà il **tipo** di un oggetto
  - ▶ 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 implementare le classi prima di creare gli oggetti
- ▶ Consente di fare **controlli di tipo statici** sugli oggetti
  - ▶ Il tipo di un oggetto sarà legato alla classe da cui è stato istanziato
  - ▶ Prende il nome di **nominal typing**

E' 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 sottotipatura 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 funzionalità

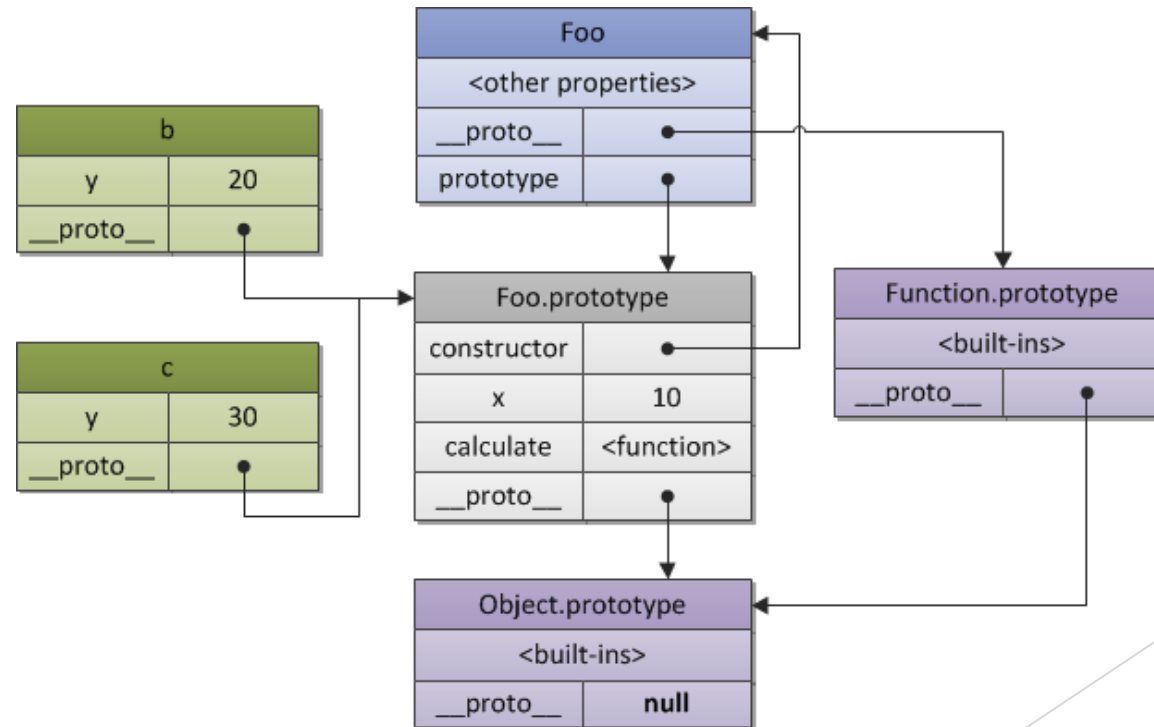
# Inheritance (Ereditarietà) - prototipi

```
// 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
```

# Inheritance (Ereditarietà) - prototipi

La gestione dei prototipi nei programmi diventa rapidamente piuttosto complicata...



# Inheritance (Ereditarietà)

- ▶ 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) - 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
```

# 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»



# 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 membri «pubblici»** (variabili e metodi utilizzabili dall'esterno... vedremo) che sono presenti anche in A
- ▶ Uno studente è sottotipo di una persona perché contiene tutto quello che c'è in persona, e anche di più
- ▶ **Corrisponde alla nozione di sottotipo già vista per i record!**

# Nominal subtyping

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

- ▶ Il **tipo di un oggetto** corrisponde alla **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 la classe B è stata definita (**sintatticamente**) come **estensione** della classe A
  - ▶ Vale la proprietà transitiva: 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** è più **flessibile**

- ▶ Non è necessario dire esplicitamente chi estende chi
- ▶ Favorisce il **polimorfismo** (la relazione di sottotipo è più debole, quindi ci sono più oggetti l'uno sottotipo dell'altro)

**Nominal Subtyping** è più **rigoroso** esplicitando i vincoli del programmatore

- ▶ Mette in relazione di sottotipo solo classi che il **programmatore** ha esplicitamente dichiarato essere in questa relazione (con extends)
- ▶ E' più **semplice da verificare** per l'interprete (deve solo controllare se una classe è nominata nella lista degli antenati dell'altra... non serve un confronto del contenuto delle due classi)

# Scelte diverse...

- ▶ Avendo una radice object-based, **JavaScript** adotta lo **structural subtyping**
- ▶ **Java** è un linguaggio class-based e adotta il **nominal subtyping**
- ▶ **OCaml** è un linguaggio in cui gli aspetti di ereditarietà 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**
- ▶ 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 dato dai **metodi** che esso contiene (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 ;;
```

# Esempio di oggetto (stack)

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

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

  (* 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>`



# REPL: uso dell'oggetto s

## NOTA SINTATTICA:

Invocazione di metodo si fa con **#-notation** invece che con dot-notation

```
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
```

# 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...

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

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

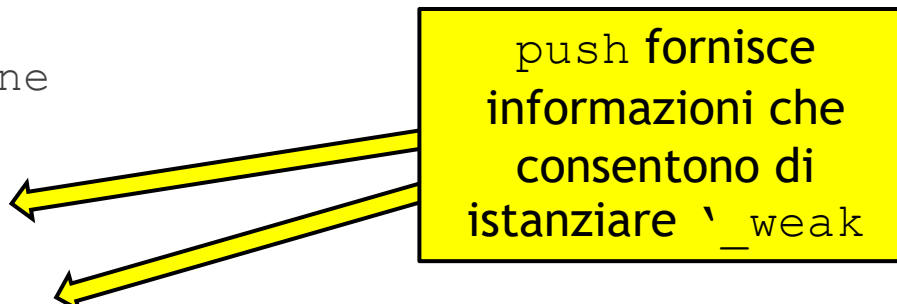
- ▶ L'oggetto *s* dovrebbe avere tipo

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

- ▶ Questo tipo concreto 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 possibile (al primo utilizzo) il tipo di *s* sarà **ricalcolato** andando ad **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 ;;  
- : int option = None  
s ;;  
val s : < pop : int option; push : int -> unit > = <obj>  
s#pop ;;  
- : int option = Some 5  
s#push "ciao" ;;  
6 | s#push "ciao" ;;  
    ^^^^^
```



push fornisce informazioni che consentono di istanziare `'_weak`

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

- Gli oggetti possono essere costruiti tramite funzioni

```
(* funzione che costruisce oggetti inizializzati con init *)  
let stack init = object  
  val mutable v = init  
  
  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] ;;
```

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

```
s#pop ;;
```

```
- : 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'istanziamento 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 sono invocati a partire dall'oggetto

► 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 si chiamano su 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 sq
```

```
val limit : < resize : int -> unit; width : int; .. > -> unit = <fun>
```

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

# Polimorfismo di oggetti

Il seguente oggetto «quadrato» può essere passato a tutte le funzioni viste

```
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**

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

esattamente come nel caso dei record. Abbiamo:

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

Grazie a cui, nel nostro esempio, possiamo derivare

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

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 contiene almeno il metodo `width`
- ▶ I puntini `..` possono essere considerati come una **variabile di tipo**  
istanziabile con una lista di metodi 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 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 >
```

E' 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 ripareremo...)



# 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` ad 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 l2 = (square 5) :: l1  
                        ^^
```

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 coercion** (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 vero 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 di oggetti**

(es. una funzione che prende oggetti **almeno** di un certo tipo)

- ▶ **principio di sostituzione**

(es. un oggetto di un tipo più specifico può essere usato ovunque serva un oggetto di un tipo più generale)

sebbene tra loro collegati, vengono **trattati in OCaml in due modi diversi** (per il secondo è richiesta 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 coercion di oggetti: OCaml VS Java

Vedremo che in **Java** sarà possibile fare delle **type coercions da un supertipo a un sottotipo** (es. trasformare uno `shape` in uno `square`)

- Questo però sarà reso possibile tramite **controlli dinamici di tipo** svolti a runtime dall'interprete della JVM (resi più facili dal **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");
```

Narrowing

# Classi

## (e costrutti linguistici per l'ereditarietà)

- ▶ Abbiamo visto che OCaml consente di lavorare direttamente con gli oggetti (in stile **object-based**)
- ▶ Uno dei meccanismi di forza della programmazione OO sono quelli legati all'**ereditarietà**
- ▶ Abbiamo visto in JavaScript che realizzare meccanismi di ereditarietà lavorando direttamente con gli oggetti richiederebbe 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
```

# Classi parametriche e polimorfe

Una classe può prevedere **parametri**

- ▶ **di costruzione** (es. `init`) che vanno passati al momento dell'istanziamento
- ▶ **di tipo** (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 ;;
```

```
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

  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 lo **structural subtyping**
- ▶ La **non modificabilità della struttura degli oggetti** (a differenza di JavaScript) consente di effettuare **controlli di tipo** a tempo di compilazione
- ▶ I costrutti **class-based** favoriscono una naturale definizione di meccanismi di **ereditarietà** (e altro...)

# OOP in OCaml: altri aspetti

OCaml prevede **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**