

# Astrazione nella programmazione

Programmazione II corso A  
Corso di Laurea in ITPS  
Università degli Studi di Bari Aldo Moro  
a.a. 2022-2023

# Astrazione e linguaggi

- L'applicazione dei principi di astrazione nella fase progettuale sarebbe poco efficace se poi nella fase realizzativa non fosse possibile continuare a nascondere l'informazione (trasformazioni, rappresentazioni, controlli di sequenza).
- Pertanto è indispensabile che i linguaggi di programmazione, che costituiscono il principale strumento utilizzato in fase di realizzazione, supportino l'applicazione dei principi di astrazione al fine di agevolare lo sviluppo e la manutenzione di sistemi software sempre più complessi.

# Principio di astrazione...

Riepilogando, negli anni 50 e 60 si introdussero diverse forme di astrazione nei linguaggi di programmazione (strutture di controllo, operatori, dati).

Si giunse quindi a maturare la convinzione che:

*“È possibile costruire astrazioni su una qualunque classe sintattica, purché le frasi di quella classe specifichino un qualche tipo di computazione”.*

# Principio di astrazione: le classi sintattiche

Di seguito si esaminerà l'applicazione del principio di astrazione a due delle seguenti classi sintattiche di seguito elencate (quelle in grassetto):

- Espressione → **astrazione di funzione**
- Comando → **astrazione di procedura**
- Controllo di sequenza → **astrazione di controllo**
- Accesso a un'area di memoria → **astrazione di selettore**
- **Definizione di un dato** → **astrazione di tipo**
- **Dichiarazione** → **astrazione generica**

**Tutte queste classi sintattiche sottintendono una computazione ...**

# ...Principio di astrazione.

Infatti, ...

- L'astrazione di una *funzione* include una espressione da valutare.
- L'astrazione di una *procedura* include un comando da eseguire.
- L'astrazione di *controllo* include delle espressioni di controllo dell'ordine di esecuzione delle istruzioni.
- L'astrazione di *selettore* include il calcolo dell'accesso ad una variabile
- L'astrazione di *tipo* include un gruppo di operatori che definiscono implicitamente un insieme di valori.
- L'astrazione *generica* include una frase che sarà elaborata per produrre legami (binding).

Controesempio:

La classe sintattica “letterale” non può essere astratta perché non specifica alcun tipo di computazione.

# Tecniche di programmazione a supporto dell'astrazione dati

- Le tecniche di **programmazione** a supporto dell'astrazione dati sono due:
  - Definizione di **tipi astratti**, cioè l'astrazione sulla classe sintattica tipo.
  - Definizione di **classi di oggetti**, cioè l'astrazione sulla dichiarazione di moduli dotati di stato locale.

# Tecniche di programmazione a supporto dell'astrazione dati

In entrambi i casi **occorre** poter **incapsulare la rappresentazione del dato con le operazioni lecite**. Tuttavia,

- Tipo astratto → il modulo rende visibile all'utilizzatore sia un identificatore di tipo che degli operatori.
- Classe di oggetti → il modulo rende visibile all'utilizzatore solo gli operatori.

Inoltre:

- Tipo astratto T → i valori sono associati a **variabili** dichiarate di tipo T.
- Classe di oggetti C → i valori sono associati a oggetti ottenuti per istanziazione della classe di oggetti C.

# Tipo concreto e tipo astratto

- I linguaggi ad alto livello mettono a disposizione del programmatore un nutrito gruppo di tipi predefiniti, detti **concreti**. Essi si distinguono in
  - *primitivi* o *semplici*: i valori associati al tipo sono atomici
  - *composti* o *strutturati*: i valori sono ottenuti per composizione di valori più semplici
- Tuttavia un linguaggio di programmazione sarà tanto più espressivo quanto più semplice sarà per il programmatore definire dei *suoi* tipi di dato a partire dai tipi di dato concreti disponibili.
  - I tipi definiti dall'utente (detti anche *user defined types*, UDT) sono anche detti **astratti**.



# Tipo concreto e tipo astratto

## Tipi concreti

(messi a disposizione  
del linguaggio)



### *Primitivi*

(valori atomici)

### *Composti*

(valori ottenuti per  
composizione di  
valori più semplici)

## Tipi astratti

(definiti dall'utente)

# Espressione di tipo (tipo)

L'***espressione di tipo*** (spesso abbreviato con ***tipo***) è il costrutto con cui alcuni linguaggi di programmazione consentono di definire un nuovo tipo. Ad esempio in *Pascal*

```
type Person = record
```

```
    name: packed array[1..20] of char;
```

```
    age: integer;
```

```
    height: real
```

```
end
```

in questo caso si

- stabilisce **esplicitamente** una **rappresentazione** per i valori del tipo *Person*
- **implicitamente** gli **operatori** applicabili a valori di quel tipo.

Per forza di cose, gli operatori del tipo *Person* dovranno essere generici (come l'assegnazione) in quanto non è possibile specificarli!

# Astrazione di tipo (Tipo Astratto)

- Una *astrazione di tipo* (o *tipo astratto di dato* o, ancora più semplicemente, *tipo astratto*) ha un corpo costituito da una espressione di tipo.
- Quando è valutata, l'astrazione di tipo stabilisce sia una rappresentazione per un insieme di valori e sia le operazioni ad essi applicabili.

l'astrazione di tipo potrà essere specificata come segue:

**type I(FP<sub>1</sub>; ...; FP<sub>n</sub>) is T**

- dove **I** è un identificatore del nuovo tipo, **FP<sub>1</sub>; ...; FP<sub>n</sub>** sono parametri formali,
- **T** è una espressione di tipo che specificherà la rappresentazione dei dati di tipo **I** e le operazioni ad esso applicabili.

# Astrazione di tipo: esempio *complex*

Analizziamo le astrazioni di tipo che alcuni linguaggi di programmazione (come C e Pascal) consentono di definire.

```
type complex = record  
           Re: real;  
           Im: real  
end
```

consente di:

1. stabilire che `complex` è un identificatore di tipo;
2. associare una rappresentazione a `complex` espressa mediante tipi concreti già disponibili nel linguaggio.

Le operazioni associate al nuovo tipo `complex` sono tutte quelle che il linguaggio ha già previsto per il tipo `record` (e.g., assegnazione e selezione di campi).

## Astrazione di tipo: limiti dell'esempio *complex*

I **limiti** di questa astrazione di tipo in Pascal sono:

1. Il programmatore **non** può definire nuovi operatori specifici da **associare** al tipo.
2. **Violazione del requisito di protezione:** l'utilizzatore
  - è consapevole della rappresentazione del tipo `complex` (sa che è un record e quali sono i campi)
  - è in grado di operare mediante operatori non specifici del dato.
3. **L'astrazione di tipo non è parametrizzata**
  - la comunicazione con il contesto esterno non è ammessa

Di seguito si mostreranno le necessarie estensioni del linguaggio per superare questi limiti.

Astrazione di tipo: definire operatori da associare al tipo

Problema: Il programmatore **non** può definire nuovi operatori specifici da **associare** al tipo.

- Il problema può essere risolto mediante un costrutto di programmazione che permetta di **incapsulare**
  1. rappresentazione del dato
  2. operatori leciti per tale rappresentazione

Questo costrutto di programmazione è il **modulo**, ovvero

- *un gruppo di componenti dichiarate come tipi, costanti, variabili, funzioni e persino (sotto) moduli.*
- si introduce quindi un costrutto **module** per supportare l'incapsulamento.

```
module complessi
```

```
  type complex= record
```

```
    R:real;
```

```
    I: real;
```

```
  end;
```

```
function RealPart(c:complex):real
```

```
  begin return(c.R) end;
```

```
function ImmPart(c:complex):real
```

```
  begin return(c.I) end;
```

```
function ConsCx(r,i:real):complex
```

```
  var c1: complex;
```

```
  begin
```

```
    c1.R := r;
```

```
    c1.I := i;
```

```
    return c1
```

```
end;
```

## Dichiarazione di tipo

```
function + (c1,c2:complex):complex
```

```
  var c3: complex;
```

```
  begin
```

```
    c3.R := c1.R+c2.R;
```

```
    c3.I := c1.I+c2.I;
```

```
  return c3
```

```
end;
```

```
function * (c1,c2:complex):complex
```

```
  var c3: complex;
```

```
  begin
```

```
    c3.R := c1.R*c2.R - c1.I*c2.I;
```

```
    c3.I := c1.R*c2.I + c1.I*c2.R;
```

```
  return c3
```

```
end;
```

```
end module.
```

## Dichiarazione di operazioni

# I Moduli

Compilazione: Il modulo introdotto è una **unità di programma compilabile separatamente**

- tutte le informazioni necessarie al compilatore per tradurre il modulo in codice oggetto sono nel modulo stesso.

Uso: per poter utilizzare le componenti dichiarate in un modulo all'interno di un'altra unità di programma (modulo o programma principale) può essere necessario esplicitare l'importazione mediante una clausola, tipo:

**uses** <nome modulo>



# I Moduli: uso

Per poter utilizzare il tipo astratto `complex` in un programma si scriverà:

`uses complessi`

In questo modo si potranno dichiarare delle variabili:

`x,y,z: complex`

e utilizzare gli operatori definiti nel modulo per i numeri complessi:

`x:=ConsCx(1.0,2.0)`

`y:=ConsCx(1.0,2.5)`

`z:=x+y`

## I Moduli: violazione del requisito di protezione

Di fatto la dichiarazione precedente considera sempre accessibili dall'esterno tutte le componenti definite nel modulo. Pertanto si potrà accedere alla rappresentazione di **complex** e scrivere

$$z.R := x.I + y.R$$

In questo modo si sta violando **il requisito di protezione: sui dati si deve poter operare solo con gli operatori definiti all'atto della specifica.**

## Moduli: soddisfacimento del Requisito di protezione

**Per soddisfare il requisito di protezione è indispensabile poter nascondere le implementazioni,** distinguendo una **parte pubblica** del modulo da una **parte privata**

- La parte pubblica contiene tutto ciò che è esportabile mediante la clausola `uses`
- La parte privata contiene le componenti del modulo non visibili dall'esterno.

In particolare le **dichiarazioni** di identificatori di tipo e i prototipi\* delle funzioni/procedure compaiono nella parte pubblica, mentre le **implementazioni** sono nella parte privata.

\* prototipo della funzione = nome della funzione + il numero e tipo dei suoi parametri

```
module complessi;  
public  
  type complex;  
  function RealPart(complex):Real;  
  function ImmPart(complex):Real;  
  function ConsCx(Real,Real):Complex;  
  ...
```

```
private  
  type complex= record  
    R: real;  
    I: real  
  end;  
  function RealPart(c:complex):real  
  begin  
    return c.R  
  end;  
  function ImmPart(c:complex):real  
  begin  
    return c.I  
  end;  
  ...  
end module.
```



# I Moduli: Parti Pubblica e Privata

- La realizzazione del tipo *complex* e dei relativi operatori è ora nascosta all'utilizzatore del modulo. **Quindi non sarà più lecito scrivere:**

$z.R := x.I + y.R$

- perché il programma che usa il modulo non sa che *complex* è di fatto realizzato come un record. La manipolazione di dati di tipo *complex* può avvenire solo attraverso gli operatori definiti nel modulo complessi.

**VANTAGGIO:** garantire il requisito di protezione significa consentire i **controlli di consistenza dei tipi** sui dati ottenuti per astrazioni, oltre che sui dati primitivi per i quali l'esecutore del linguaggio assicura già tali controlli. **Quindi tutti i dati, siano essi primitivi o astratti, sono trattati in modo uniforme.**

# Moduli: operatori privati

Finora tutte le operazioni definite nel modulo sono state considerate accessibili dall'esterno. Tuttavia, quando la realizzazione di tali operazioni risulta complessa, si potrebbe voler scrivere dei **sottoprogrammi** utilizzabili nella realizzazione delle operazioni ma **non accessibili dall'esterno**. Questi sono detti **privati** e compaiono solo nella parte privata del modulo.

```
module razionali;  
public  
type Rational;  
const Rational zero;  
const Rational one;  
function ConsRat(Integer,Integer):Rational;  
function +(Rational,Rational):Rational;  
function =(Rational,Rational):Boolean;
```

```
private
  type Rational = record
    N: Integer;
    D: Integer
  end;

  const Rational zero=(0,1);
  const Rational one=(1,1);
  function Prod(a,b,c,d: Integer):Integer;
    begin
      return ( a*d + b*c );
    end
  ...
  function +(r1,r2: Rational):Rational
  var r3: Rational;
  begin
    r3.N := prod(r1.N,r1.D,r2.N,r2.D);
    r3.D := r1.D*r2.D;
    return r3
  end
  ...
end module.
```

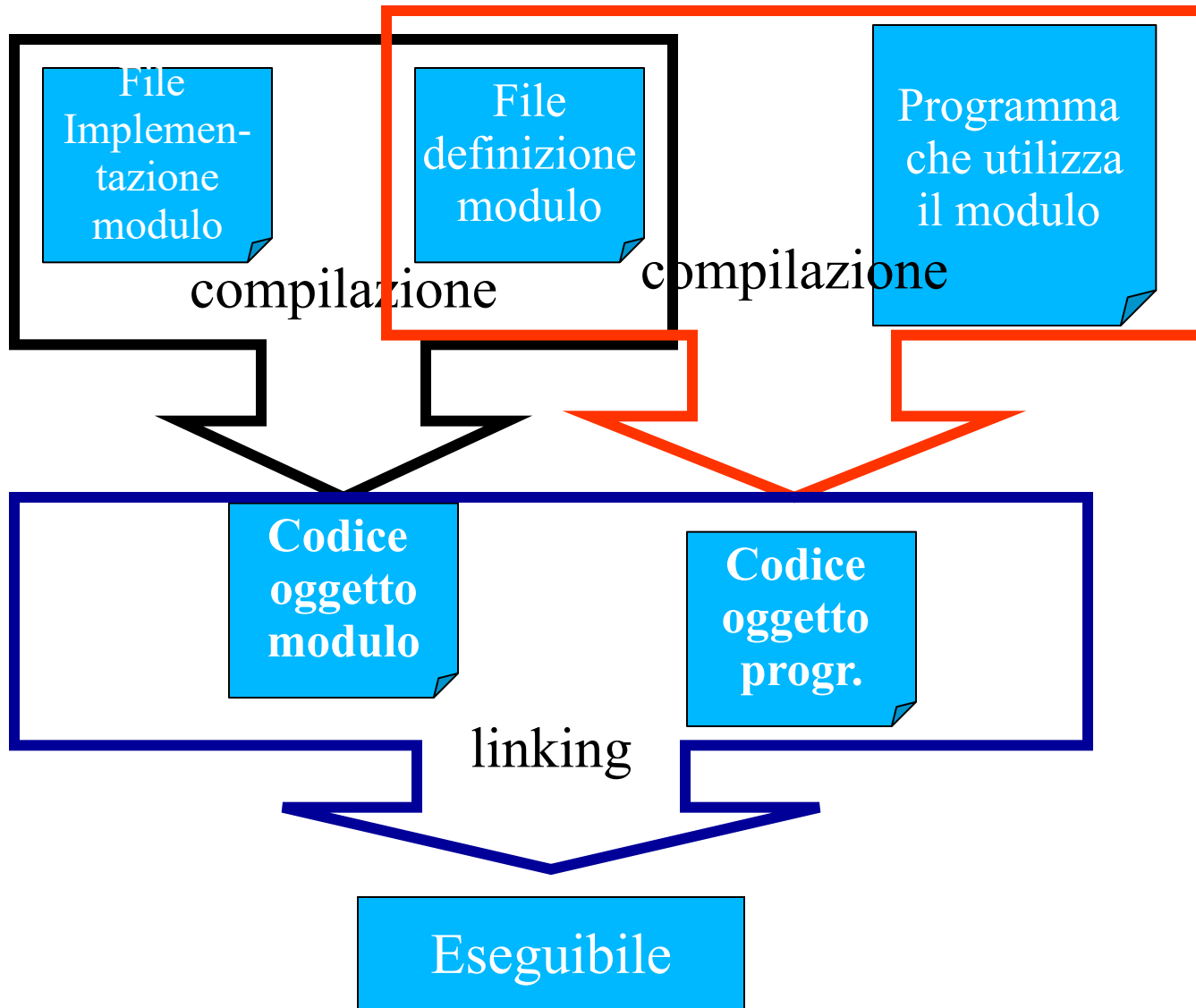
il prototipo della  
funzione non è presente  
nella sezione **public**

# I Moduli: distribuzione in file

- Operativamente, ci si potrebbe aspettare che **la parte pubblica di un modulo e quella privata finiscano in due file differenti** (file di definizione e file di implementazione).
  - Il compilatore necessita di entrambi i file per produrre il codice oggetto del modulo.
  - Il compilatore dovrebbe aver bisogno solo del file di definizione per produrre il codice oggetto di un programma che importa il modulo.
  - I due moduli oggetti dovrebbero poi essere collegati (fase di link) per produrre il file eseguibile.



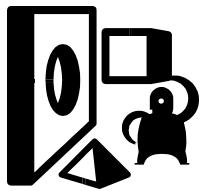
# I Moduli: Distribuzione in file



# I Moduli: Distribuzione in file

**Vantaggio della separazione:** è possibile distribuire il file di definizione insieme al codice oggetto del modulo senza fornire il file di l'implementazione.

Tuttavia si pone un ...



**Problema:** consideriamo il modulo *complessi* e la seguente dichiarazione presente nel programma che lo importa:

*x,y,z: complex*

Essa non può essere compilata, perché non è nota la struttura di un dato di tipo complex e **non si sa quanto spazio di memoria riservare per le tre variabili.**

# I Moduli: Distribuzione in file

## **Soluzione 1:** uso dei puntatori

- Ricorrere ai tipi *opachi* del Modula-2, cioè a tipi dichiarati nella parte pubblica ma definiti in quella privata, che devono essere obbligatoriamente **di tipo puntatore**.
  - In questo modo il compilatore sa che ad ogni variabile dichiarata come tipo opaco dev'essere riservato lo spazio tipicamente riservato a un puntatore.

Quello che si nasconde effettivamente è la struttura dell'oggetto puntato, oltre alla implementazione degli operatori.

# I Moduli: Distribuzione in file

**Soluzione 2:** file di definizione con definizioni pubblicate e private

- il file di definizione include sia le dichiarazioni pubbliche di tipi, costanti, funzioni e procedure e sia le definizioni private dei tipi e delle costanti.
- nel file di implementazione sono riportate solo le implementazioni delle procedure e delle funzioni.
  - In questo modo l'informazione sulla dimensione di un dato *complex* è nota al compilatore. Tuttavia il compilatore non utilizzerà mai l'informazione sulla struttura per permettere l'accesso ai dati il cui tipo è dichiarato come privato.

# I moduli con stato locale: tipi astratti

- I moduli che contengono solo definizioni di tipi, procedure e funzioni non hanno un proprio stato permanente (sono *stateless*).
  - Le variabili definite nelle procedure e nelle funzioni del modulo hanno un tempo di vita pari a quello dell'esecuzione della procedura o funzione che le contiene.
- Si può pensare di estendere la definizione di modulo, consentendo anche la *definizione di variabili esterne alle procedure e funzioni*.
  - Tali variabili avranno un tempo di vita pari a quello dell'unità di programma che “usa” il modulo.

# I moduli con stato locale: gli oggetti

In questo modo il modulo viene dotato di un suo *stato locale*, che consiste nel valore delle variabili esterne a procedure e funzioni. Un modulo dotato di stato locale è chiamato *oggetto*.

Il termine *oggetto* è usato anche per la variabile nascosta del modulo.


Le funzioni e le procedure incapsulate in un oggetto sono comunemente chiamate *metodi*.

# I moduli con stato locale: esempio

```
module stack10real;
public
  procedure push(real);
  function pop():real;
private
  type stack_object=
    record
      st:array[1..10] of real;
      top: 0..10;
    end;
  var s:stack_object;

  procedure push(elem:real)
  begin
    if s.top<10 then
      begin
        s.top:= s.top+1;
        s.st[s.top]:=elem;
      end;
    end;
  end;

  function pop():real;
  begin
    if s.top>0 then
      begin
        s.top := s.top-1;
        return s.st[s.top+1];
      end;
    end;
  end;
end module.
```



A diagram consisting of a horizontal line from the `end;` of the `push` procedure, followed by a vertical line that turns right into an arrow pointing to the `function pop()` definition.

# I moduli con stato locale

Il tempo di vita della variabile **s** è quello del modulo.

- La variabile è usata ripetutamente in chiamate successive degli operatori `push` e `pop`.
- Il suo valore condiziona anche l'ordine di chiamata delle procedure, per cui non è possibile invocare la funzione `pop` se non dopo una `push` che ne modifica il campo `top`.

**Problema:** come ci si può assicurare che `top` sia inizializzata precedentemente alla prima invocazione della procedura `push`?



# I moduli con stato locale

Ovviamente **per `top`** basterebbe una dichiarazione con **inizializzazione**.

Più in generale, per poter inizializzare l'intera struttura **s** si potrebbero adottare due soluzioni:

1. Dotare il modulo di una procedura **`init()`** da invocare dopo aver importato il modulo.
2. Dotare il modulo di un ***main***, eseguito solo nel momento in cui il modulo è importato in altri moduli mediante “uses”.

La seconda soluzione è preferibile perché non si basa sul corretto utilizzo del modulo da parte del programmatore.


# I moduli con stato locale

## Esempio

```

module stack10real;
public
  procedure push(real);
  function pop():real;
private
  type stack_object=
    record
      st:array[1..10] of real; end;
      top: 0..10;
    end;
  var s:stack_object;
  ...
  procedure push(elem:real)
  begin
    if s.top<10 then
      begin
        s.top:= s.top+1;
        s.st[s.top]:=elem;
      end;
    end;
  end;
end;

```

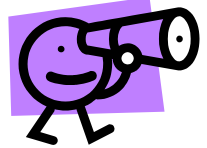


```

function pop():real;
begin
  if s.top>0 then
    begin
      s.top := s.top-1;
      return s.st[s.top+1];
    end;
  begin
    integer i;
    s.top:=0;
    for i:=1 to 10 s.st[i]:=0;
  end;
end module.

```

# I moduli con stato locale



Osservazioni:

1. La definizione del tipo **stack\_object** è ora privata.
  - l'utente può solo usare l'oggetto **stack10real** e non può dichiarare variabili di tipo **stack\_object** nell'unità di programma che importa il modulo.
2. È possibile cambiare lo stato locale solo attraverso gli operatori **push** e **pop**, che costituiscono l'**interfaccia** dell'oggetto con il mondo esterno.
3. Il modulo, mediante le proprie variabili esterne a funzioni e procedure, può:
  - controllare l'ordine di accesso alla variabile nascosta, ad esempio, per impedire l'estrazione dei dati prima che la variabile venga avvalorata.

## I moduli con stato locale: allocazione in memoria

Quando una unità di programma **P** importerà il modulo **stack10real** con il comando:

**uses stack10real**

allora verranno allocati in memoria

- un array di 10 reali **st**
- una variabile intera **top**

che saranno comunque inaccessibili a **P**.

Inoltre verrà eseguito il main del modulo **stack10real** che provvederà a inizializzare **top** e gli elementi di **st** a zero.

# I moduli con stato locale

Così le chiamate alla procedura

**push(elem)**

e alla funzione

**pop()**

modificheranno **top** e **st** senza che ciò sia osservabile dall'esterno.

## I moduli con stato locale: uso di moduli analoghi

Va osservato che in questo modo il programma **P** ha accesso ad un solo oggetto “stack di 10 valori reali”. Se ce ne fosse bisogno di un altro, potremmo scrivere un altro modulo, denominato **stack10real2**, analogo al precedente.

Tuttavia, in questo modo sorgerebbe il problema di come distinguere gli operatori `push` e `pop`, ora ambigui.



***Soluzione:*** usare la notazione puntata per riferirsi al modulo appropriato sul quale invocare la procedura o la funzione.

## I moduli con stato locale: notazione puntata

### Esempio

**Stack10real.push(elem)**

**Stack10real.pop()**

**Stack10real2.push(elem)**

**Stack10real2.pop()**

# Moduli generici: Classi di oggetti

In generale, un oggetto è un insieme di variabili interne ad un modulo e manipolabili esternamente solo mediante gli operatori (pubblici) definiti nel modulo stesso.

Da quanto visto finora, per poter definire più oggetti “simili” o “dello stesso tipo”, cioè con medesima rappresentazione e stesso insieme di operatori, si è costretti a definire tanti moduli quanti sono gli oggetti che si vogliono usare nel programma. Tutti questi moduli differiranno solo per l’identificatore del modulo (*identificatore dell’oggetto*).

Per evitare l’inconveniente di dover duplicare un modulo si può pensare di definire un *modulo generico* che identifica una *classe* di oggetti simili. I singoli oggetti sono poi ottenuti con il meccanismo della *istanziatura* della classe.




# Classi di oggetti

## Esempio

```

generic module stack10real;
public
    procedure push(real);
    function pop():real;
private
    type stack_object=
        record
            st:array[1..10] of real;
            top: 0..10;
        end;
    var s:stack_object;
    ...
    procedure push(elem:real)
    begin
        if s.top<10 then
            begin
                s.top:= s.top+1;
                s.st[s.top]:=elem;
            end;
        end;
    end;
end;

```



```

function pop():real;
begin
    if s.top>0 then
        begin
            s.top := s.top-1;
            return s.st[s.top+1];
        end;
    end;
begin
    integer i;
    s.top:=0;
    for i:=1 to 10 s.st[i]:=0
end;
end module.

```

# Classi di oggetti: istanziazione

Continuando a estendere un ipotetico linguaggio di programmazione, una istanziazione potrebbe essere ottenuta mediante un costrutto di questo genere:

- `st1 instantiation of stack10real;`
- `st2 instantiation of stack10real;`

Le istanziazioni creano due oggetti simili di tipo stack denotati rispettivamente `st1` e `st2`. Al momento della istanziazione verranno allocati in memoria

- due array di 10 real
- due variabili intere

Inoltre verrà eseguito due volte il main del modulo generico per inizializzare a zero i puntatori al top degli stack e gli elementi stessi degli stack.

# Classi di oggetti: accesso agli operatori

Agli oggetti si potrà accedere mediante gli operatori PUSH e POP, ma con l'accorgimento di far precedere il nome dell'operatore dall'identificatore dell'oggetto, altrimenti il compilatore non potrebbe risolvere correttamente i riferimenti per via della loro definizione in entrambi i moduli **st1** e **st2**.

Quindi scriveremo:

```
st1.push(10)
```

```
m := st2.pop()
```

# Astrazione della dichiarazione di modulo

- La precedente definizione di una **classe** corrisponde a una particolare forma di astrazione, quella della classe sintattica '*dichiarazione di un modulo*'.
- L'operazione di istanziiazione corrisponde alla invocazione di questa astrazione ed ha l'effetto di 'creare legami' (binding). In particolare, si crea un legame fra
  - Identificatore dell'oggetto
  - Nome della classe (cioè nome del modulo generico)
- la definizione di una classe corrisponde a una particolare forma di **astrazione generica**, cioè di astrazione applicata alla classe sintattica dichiarazione.

# Astrazione della dichiarazione di modulo

- Quando si affronterà l'argomento dell'astrazione generica in modo sistematico, si osserverà che è possibile astrarre anche su altre dichiarazioni (per esempio, funzioni e procedure), oltre quella di modulo.

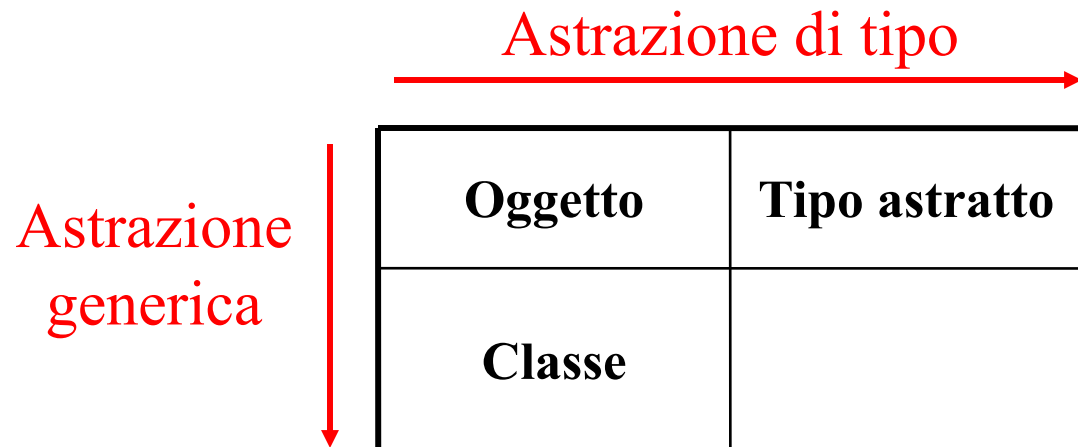


**Riflessione:** si è già visto che un modulo può essere usato per incapsulare un **tipo astratto** con gli operatori leciti. Ha senso definire *generic* un modulo di questa fatta e poi istanziarlo?

# Tipi astratti o classi?

Riepilogando, se in fase di progettazione si identifica l'esigenza di disporre di un dato astratto, in fase realizzativa si può:

- 1) **Definire un oggetto**: la scelta è appropriata nel caso in cui si necessiti di una sola occorrenza del dato astratto;
- 2) **Definire un tipo astratto**: l'astrazione riguarda la classe sintattica tipo;
- 3) **Definire una classe**: l'astrazione riguarda la dichiarazione di un modulo (dotato di stato locale).



# Tipi astratti o classi?

In tutti i casi la rappresentazione del dato astratto viene nascosta e la manipolazione dei valori è resa possibile solo mediante operazioni fornite allo scopo.

Tuttavia ci sono delle **differenze** fra tipo astratto e classe di oggetti ...

# Tipi astratti o classi?

- *Sintattica*: nel tipo astratto gli operatori hanno **un parametro in più**, relativo proprio al tipo che si sta definendo.

Esempio: nel tipo astratto Stack, gli operatori POP e PUSH hanno un parametro in più di tipo Stack che gli operatori definiti per la classe Stack non hanno.



# Tipi astratti o classi?

- *Concettuale*: richiamando la suddivisione delle operazioni su un dato astratto in osservazioni e costruttori (vedi specifiche algebriche), si può dire che:
  - il tipo astratto è **organizzato intorno alle osservazioni**. Ogni osservazione è implementata come una operazione su una rappresentazione concreta derivata dai costruttori. Anche i costruttori sono implementati come operazioni che creano valori. La rappresentazione è **condivisa** dalle operazioni, ma è nascosta ai fruitori del tipo astratto.
  - La classe è **organizzata intorno ai costruttori**. Le osservazioni diventano metodi dei valori. Un oggetto è definito dalla combinazione di tutte le osservazioni possibili su di esso.

# Svantaggi del tipo astratto

- *Scarsa estendibilità*: l'aggiunta di un nuovo costruttore comporta dei cambiamenti intrusivi nelle implementazioni esistenti degli operatori.

Ogni operatore dovrà essere opportunamente rivisto in modo da prevedere il trattamento di rappresentazioni ottenute con nuovi costruttori.

Esempio: si vuole implementare il dato astratto *geometricShape* la cui specifica algebrica è fornita di seguito:

<i>osservazioni</i>	<i>Costruttore di g</i>	
	<i>square(x)</i>	<i>circle(r)</i>
<i>area(g)</i>	$x^2$	$\pi r^2$

# Definizione tipo astratto FormaGeometrica

Esempio (cont.): Definendo un **tipo astratto** *geometricShape* occorrerà scrivere un modulo che definisca tale tipo ed includa tre funzioni:

*square(x: real): geometricShape*                      crea un quadrato di lato x;

*circle(r: real): geometricShape*                      crea un cerchio di raggio r;

*area(g: geometricShape): real*                      calcola l' area

Infatti, **tutte le operazioni che hanno necessità di operare sulla rappresentazione di *geometricShape* devono stare nel modulo dove essa è definita (raggruppamento intorno alle operazioni).**

Una possibile rappresentazione in Pascal per *geometricShape* sarà la seguente:

```
type geometricShape = record
                                shape: char;
                                value: real
                            end
```

Una possibile realizzazione dei tre operatori è:

# Funzioni FormaGeometrica

Esempio (cont.):

function <i>square(x: real): geometricShape</i>	function <i>circle(x: real): geometricShape</i>
var g: geometricShape	var g: geometricShape
begin	begin
g.shape := 's' ;	g.shape := 'c' ;
g.value := x;	g.value := x;
return g	return g
end	end

```
function area(g: geometricShape): real
begin
  if g.shape = 's'
    then return g.value * g.value
  else
    return 3.14 * g.value * g.value;
  end
```

# Forma geometrica realizzata con le classi

Domanda: Cosa accadrebbe se Forma Geometrica fosse realizzata con le classi?

<i>osservazioni</i>	<i>Costruttore di g</i>	
	<i>square(x)</i>	<i>circle(r)</i>
<i>area(g)</i>	$x^2$	$\pi r^2$

Le classi sono costruite attorno ai costruttori!

Ogni costruttore è convertito in una classe per costruire gli oggetti

- ogni costruttore darà origine a 2 oggetti diversi per ognuno dei quali saranno definite delle classi

# FomaGeometrica mediante classi

Realizzando il dato astratto mediante **classi** si possono definire due moduli generici, uno per ogni forma geometrica (**o costruttore**).

```
generic module Circle
public
    function area(): real;
    procedure init(real);
private
    var raggio:real;
procedure init(x:real)
begin
    raggio := x
end;
function area():real
begin
    return 3.14*raggio*raggio
end;
end module.
```

```
generic module Square
public
    function area(): real;
    procedure init(real);
private
    var lato:real;
procedure init(x:real)
begin
    lato := x
end;
function area():real
begin
    return lato*lato
end;
end module.
```

Per utilizzare una forma geometrica si deve istanziare una delle due classi e invocare il metodo *init*.

# Estensione di FormaGeometrica: tipi astratti

Cosa accadrebbe se si estendesse la specifica del dato astratto in modo da considerare anche i rettangoli?

<i>osservazioni</i>	<i>Costruttore di g</i>		
	<i>square(x)</i>	<i>circle(r)</i>	<i>rectangle(l,m)</i>
<i>area(g)</i>	$x^2$	$\pi r^2$	$l \cdot m$

Se abbiamo specificato un **tipo astratto**, siamo stati costretti a cambiare la rappresentazione in modo da memorizzare due valori (per i due lati del rettangolo) e non uno.

**type geometricShape = record**

**shape: char;**

**value: real;**

**value2: real**

**end**

# Svantaggi del tipo astratto

Inoltre dobbiamo aggiungere l'opportuno costruttore ...

```
function rectangle(x,y: real): geometricShape
```

```
var g: geometricShape
```

```
begin
```

```
  g.shape := 'r' ;
```

```
  g.value := x;
```

```
  g.value2 := y;
```

```
  return g
```

```
end
```

... e dobbiamo modificare la  
anche la funzione area ☹

```
function area(g: geometricShape): real
```

```
begin
```

```
  if g.shape = 's' then return g.value * g.value
```

```
    else if g.shape = 'c' then return 3.14 * g.value * g.value
```

```
    else return g.value * g.value2
```

```
end
```

Sempre supponendo di disporre del codice sorgente del modulo! 56



# Estensione di FormaGeometrica: classi

Diversamente, avendo realizzato il dato astratto mediante **classi** basta aggiungere un'altra classe.

```
generic module Rectangle
public
  function area(): real;
  procedure init(real,real);
private
  var base, altezza:real;
procedure init(x,y:real)
begin
  base := x;
  altezza := y
end;
function area():real
begin
  return base*altezza
end;
end module.
```

Per utilizzare un rettangolo si dovrà istanziare questa classe e invocare il metodo *init*.

Nella programmazione orientata a oggetti, per evitare la riscrittura di codice comune ad altre classi già definite è possibile ricorrere ai meccanismi di **ereditarietà** tra le classi.

# Riferimenti bibliografici

M. Shaw

Abstraction Techniques in Modern Programming Languages

*IEEE Software*, 10-26, October 1984.

D. A. Watt

*Programming Language Concepts and Paradigms (cap. 5-6)*

Prentice Hall, 1990.

W.R. Cook

Object-Oriented Programming Versus Abstract Data Types

In J.W. de Bakker et al., editor, *Foundations of Object-Oriented Languages*, number 489 in Lecture Notes in Computer Science, pagine 151–178. Springer, 1991.

B. Meyer

Genericity vs. Inheritance

*Proceedings OOPSLA '86*, pp. 391-405