

# Il paradigma Orientato agli Oggetti: nozioni di base

versione del 18/10/2022

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

# Sommario

- Il paradigma di programmazione Object Oriented (OO)
- Le basi del paradigma OO (cenni)
  - Astrazione (ampiamente trattata in «Astrazione nella progettazione» e in «astrazione nella programmazione»)
  - Modularità
  - Protezione (Information Hiding)
  - Ereditarietà
  - Polimorfismo (di inclusione e parametrico)
  - Dinamicità
- Gli oggetti e la loro identità

# Introduzione


# I Paradigmi di programmazione ...

- La storia dell'informatica è stato un lento ma costante attraversamento dello spettro **what-how**.
- La scienza della programmazione ha esplorato molti punti dello spettro, sempre combattuta fra il “cosa” e il “come”.
- Ne sono nati diversi **paradigmi di programmazione**, ovvero *collezioni di modelli concettuali che insieme plasmano il processo di analisi, progettazione e sviluppo di un programma*.


**Paradigma**: dal greco παραδειγμα ‘modello, esempio’, da παρα ‘presso, accanto’ (indica similarità) + δεικνυναι ‘mostrare, indicare’

## ...I paradigmi di programmazione ...

- Questi modelli concettuali “strutturano” il pensiero in quanto determinano la forma di programmi validi.
- Essi influenzano il modo in cui pensiamo e formuliamo le soluzioni, arrivando a condizionare perfino la possibilità di trovare una soluzione.



Per calcolare il fattoriale di  $n$  moltiplica 1 per 2, poi per 3, ... fino ad arrivare a  $n$



Il fattoriale di  $n$  è  $n$  per il fattoriale di  $n-1$

## ...I paradigmi di programmazione

- Un paradigma cambia fundamentalmente il modo in cui guardiamo ai problemi incontrati nel passato.
- Esso ci deve dare un nuovo schema per pensare ai problemi futuri.
- Esso cambia le nostre priorità, le nostre idee su quanto merita attenzione e su cosa sia importante.

# Rapporto Paradigma-Linguaggi

Nel senso della macchina di Turing, tutti i linguaggi di programmazione più comuni sono universali.

Tuttavia ogni linguaggio di programmazione si basa, o meglio *supporta*, un particolare paradigma, fornendo:

- a) Le *primitive* di quel paradigma.
- b) I *metodi di composizione* di quel paradigma.
- c) Un *linguaggio utente appropriato* che rende chiari i programmi scritti secondo quel paradigma
- d) Una *esecuzione efficiente* di programmi scritti in quello stile.

# Rapporto Paradigma-Linguaggi

I linguaggi di programmazione sono, dunque, dotati di opportuni *costrutti linguistici* che riflettono i modelli concettuali di un paradigma, al fine di facilitare l'espressione di una soluzione definita attraverso i modelli concettuali del paradigma.

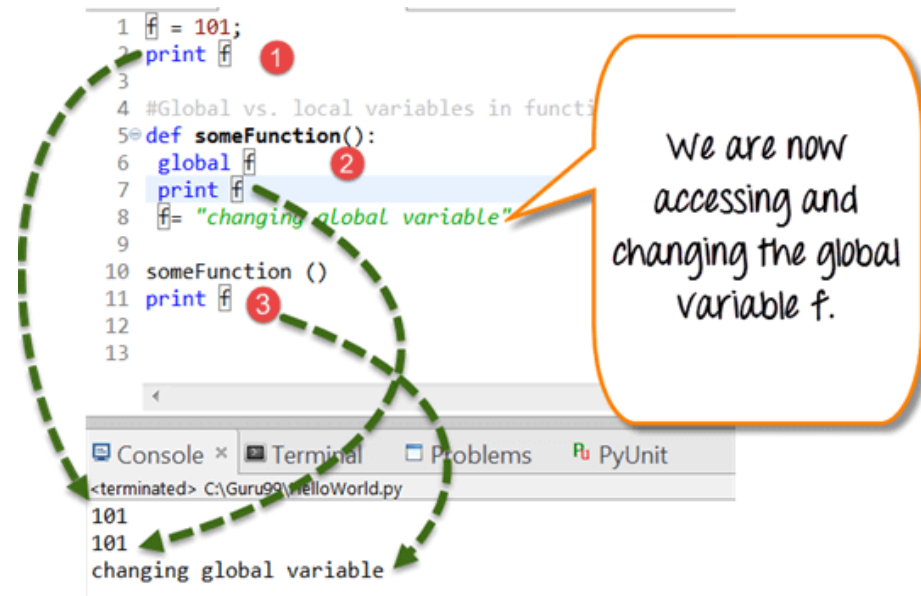
In realtà, i linguaggi di programmazione possono supportare *più di un paradigma*.



# Perché cambiare paradigma?

## Il problema delle variabili globali

- Un difetto fondamentale della programmazione imperativa è che le variabili globali sono potenzialmente accessibili da ogni parte del programma.
- I grandi programmi che difettano di una disciplina di accesso alle variabili globali tendono ad essere ingestibili.
  - nessun modulo che accede ad una variabile globale può essere sviluppato e compreso indipendentemente da altri moduli che pure accedono alla medesima variabile.



# Gli oggetti e la programmazione imperativa

Nella programmazione imperativa è possibile definire degli oggetti, tuttavia

- il loro utilizzo è legato all'auto disciplina dei programmatori e non viene in alcun modo forzato
  - Non ci sono meccanismi nativi a supporto della definizione ed all'uso degli oggetti

## *Il paradigma OO: una rivoluzione*

Il paradigma orientato a oggetti rispetto a quello imperativo costituisce:

- una ***rivoluzione***, in quanto gli oggetti assumono un ruolo fondamentale nella progettazione e nella programmazione.

**Information hiding** + **incapsulamento** sono principi cardine nel paradigma orientato a oggetti

# La classificazione di Wegner (1987\*)

I linguaggi di programmazione *object-native* si classificano in:

- Object-based
  - supportano la nozione di oggetto (Modula-2)
- Class-based
  - supportano la nozione di oggetto e classe (Ada-83)
- Object-oriented
  - supportano la nozione di oggetto, classe, ereditarietà (Smalltalk, C++, Java, ...)

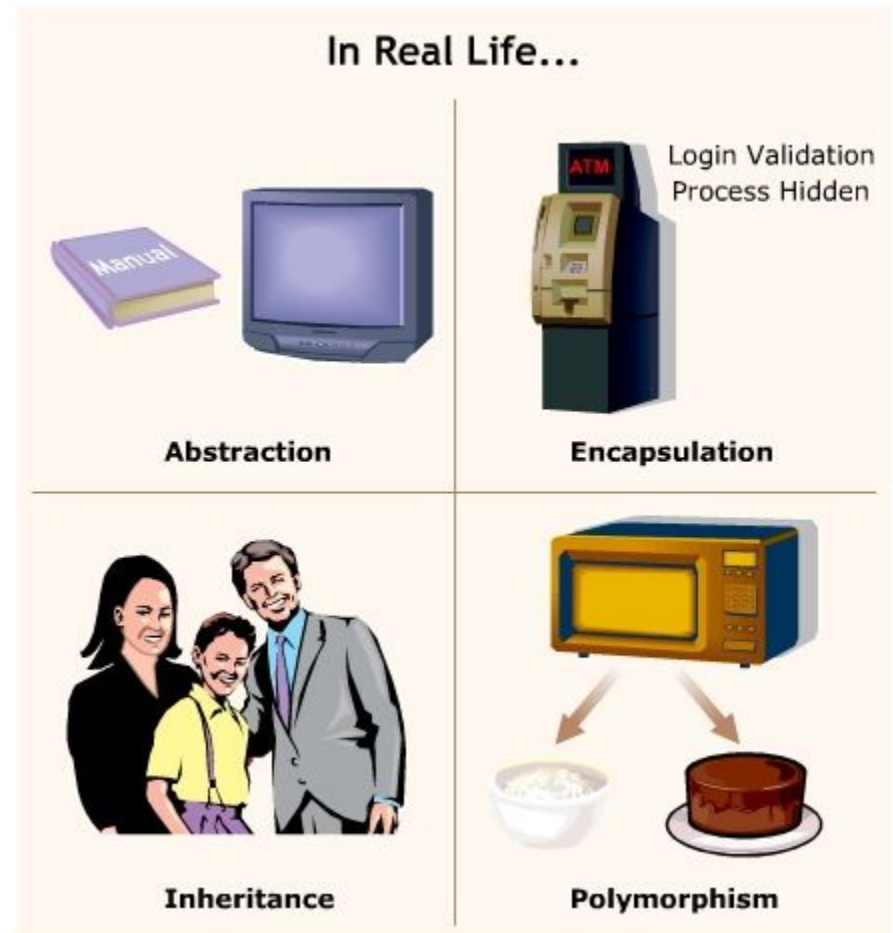
Le caratteristiche fondamentali del paradigma di programmazione OO mediante la seguente equazione sono:

$$\textit{object-oriented} = \textit{objects} + \textit{classes} + \textit{inheritance}$$

\*Peter Wegner. 1987. Dimensions of object-based language design. In Conference proceedings on Object-oriented programming systems, languages and applications (OOPSLA '87), Norman Meyrowitz (Ed.). ACM, New York, NY, USA, 168-182. DOI=<http://dx.doi.org/10.1145/38765.38823>

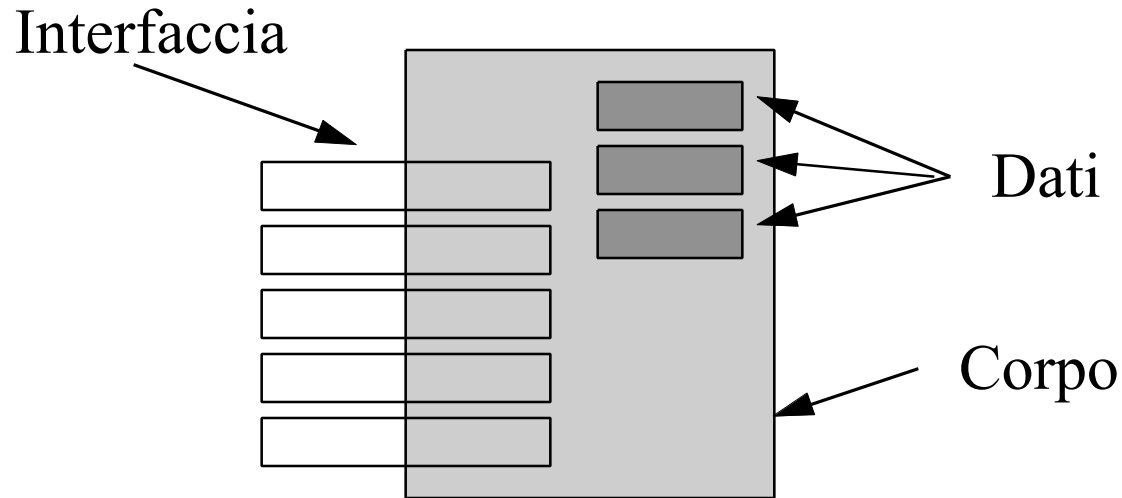
# Concetti basici dei linguaggi OO

- Astrazione
  - Modularità
  - Protezione (Information Hiding)
- Ereditarietà
- Polimorfismo (vari tipi)
- Dinamicità
- Oggetto, Classe



# Modularità e protezione

- Un modulo consiste di un *interfaccia* e di una *implementazione*



- Chi usa il modulo conosce solo l'interfaccia
- L'implementazione è *nascosta*

# Modularità

- Tre categorie di moduli
  - Librerie: Moduli che offrono solo funzioni e procedure e non contengono dati
  - Strutture Dati Astratte (ADS, astrazioni di dato): una ADS consiste di dati e funzioni: i dati sono *nascosti* e possono essere modificati solo dalle funzioni del modulo
  - Tipi di Dati Astratti (ADT): un ADT è simile a una ADS, ma rappresenta un tipo di dato

# Information Hiding

- David Parnas suggerì di incapsulare in un modulo (**Information Hiding**) ogni variabile globale insieme a un gruppo di operazioni autorizzate ad accedervi. Gli altri moduli possono accedere alla variabile solo indirettamente, chiamando queste operazioni.
- L'idea di Parnas era proprio quella di definire degli *oggetti*

Osservazione: Definizione di IH complementare a quella fornita nella dispensa «Astrazione nella progettazione»



# Obiettivo dell'Information Hiding

Rendere possibile **il cambiamento del comportamento** di un modulo **senza:**

- senza modificare il comportamento degli altri moduli
- conoscere il comportamento degli altri componenti

# Ereditarietà

- Meccanismo per costruire nuovi ADT partendo da quelli esistenti *ereditandone* la definizione
  - Favorisce il riuso del codice (principio di riusabilità)
  - Favorisce l'estendibilità del codice (progettazione per estensione e non per modifica)

# Ereditarietà: esempio vagone

## senza ereditarietà

Problema: definire i vagoni di un treno

Box car



Tank car



Engine



Caboose



Realizzare una soluzione che permetta di tenere traccia di 4 differenti tipi di vagoni merci:

- Box Car. Dotato di volume, percentuale di carico, anno di costruzione;
- Tank Car. Dotato di volume, percentuale di carico, anno di costruzione;
- Engine. Dotato di anno di costruzione;
- Caboose: Dotato di anno di costruzione

Requisito r1: per ogni tipo di vagone al momento della creazione deve essere specificato l'anno di costruzione

Considerazioni utili alla progettazione della soluzione:

- percentuale di carico ricorre in Box Car e Tank Car ed è calcolata nello stesso modo
- volume ricorre in Box Car e Tank Car ed è calcolata in modi differenti
- anno di costruzione ricorre in tutti i tipi di vagoni

# Ereditarietà: vagone

## soluzione senza ereditarietà: BoxCar

**generic** module BoxCar;

**public**

**function** volume(): real;

**function** age(): integer

**function** percent\_loaded(): integer

**costruttore**(heightP, widthP, lengthP,  
year\_builtP)

**private**

integer height=0, width=0, length=0,  
year\_built=0;

integer percent\_loaded=0;

**function** volume(): real;

begin

return height\*width\*length;

end

– **function** age(): integer;

- begin

- Date date instantiation of Date;

- return date.getCurrentDate()-year\_built;

- end

– **function** percent\_loaded(): integer;

- begin

- // some instructions to calculate tank  
percentage loaded

- end

– **begin** (heightP, widthP, lengthP,  
year\_builtP)

- height= heightP;

- width= width P;

- length= lengthP;

- year\_built=yearBuiltP;

– **end**

**end module;**

# Ereditarietà: vagone

## soluzione senza ereditarietà: TankCar

```
uses Math.PI;
generic module TankCar;
public
    function volume(): real;
    function age(): integer
    function percent_loaded(): integer
    costruttore(heightP, widthP, lengthP,
        year_builtP)
private
    integer radius=0, year_built=0;
    integer percent_loaded=0;
    function volume(): real;
        begin
            return Math.PI*radius*radius;
        end
```

```
function age(): integer;
    begin
        Date date instantiation of Date;
        return date.getCurrentDate()-year_built;
    end
function percent_loaded(): integer;
    begin
        // some instructions to calculate tank
        percentage loaded
    end
begin (radiusP, year_builtP)
    radius= radiusP;
    year_built=yearBuiltP;
end
end module;
```

# Ereditarietà: vagone

soluzione senza ereditarietà: EngineCar, CabooseCar

generic module EngineCar;

public

function age(): int;

costruttore(year\_builtP);

private

integer year\_built=0;

function age(): int;

begin

Date date instantiation of Date;

return date.getCurrentDate()-year\_built;

end

begin (year\_builtP)

year\_built=yearBuiltP;

end

end module;

generic module CabooseCar;

public

function age(): int;

costruttore(year\_builtP);

private

integer year\_built=0;

function age(): int;

begin

Date date instantiation of Date;

return date.getCurrentDate()-year\_built;

end

begin (year\_builtP)

year\_built=yearBuiltP;

end

end module;

# Ereditarietà: esempio vagone

## con ereditarietà

Problema: definire i vagoni di un treno

Realizzare una soluzione che permetta di tenere traccia di 4 differenti tipi di vagoni merci:

- Box Car. Dotato di volume, percentuale di carico, anno di costruzione;
- Tank Car. Dotato di volume, percentuale di carico, anno di costruzione;
- Engine. Dotato di anno di costruzione;
- Caboose. Dotato di anno di costruzione

Requisito r1: per ogni tipo di vagone al momento della creazione deve essere specificato l'anno di costruzione

Box car



Tank car



Engine



Caboose



### Considerazioni utili alla progettazione della soluzione:

- percentuale di carico ricorre in Box Car e Tank Car ed è calcolata nello stesso modo
- volume ricorre in Box Car e Tank Car ed è calcolata in modi differenti
- anno di costruzione ricorre in tutti i tipi di vagoni

# Ereditarietà: vagone

## soluzione con ereditarietà

*/\* fattorizza gli elementi comuni di Box Car e Tank Car \*/*

**generic module** Container;

**public**

**function** percent\_loaded(): integer

**function** volume(): real;

**private**

integer percent\_loaded=0;

**function** percent\_loaded(): integer;

**begin**

*// some instructions to calculate tank  
percentage loaded*

**end**

**end module;**

*/\* per la funzione volume() è definito solo il prototipo, la  
realizzazione differirà in base al tipo di Container concreto \*/*

*/\* fattorizza tutti gli elementi comuni a tutti i  
tipi di vagoni \*/*

**generic module** RailroadCar;

**public**

**function** age(): int;

*costruttore*(year\_builtP)

**private**

integer year\_built=0;

**function** age(): int;

**begin**

*Date date instantiation of Date;*

*return date.getCurrentDate()-year\_built;*

**end**

*// requisito r1 soddisfatto dal costruttore*

**begin** (year\_builtP)

*year\_built=yearBuiltP;*

**end**

**end module;**



# Ereditarietà: vagone

## soluzione con ereditarietà: EngineCar, CabooseCar

```
// inizio modulo EngineCar
```

```
generic module EngineCar inherits  
RailroadCar;
```

```
public
```

```
    costruttore(year_builtP);
```

```
private
```

```
    begin (year_builtP)
```

```
        costruttoreClasseSuper(year_builtP)
```

```
    end
```

```
end module // fine modulo EngineCar
```

```
// inizio modulo CabooseCar
```

```
generic module CabooseCar inherits  
RailroadCar;
```

```
public
```

```
    costruttore(year_builtP);
```

```
private
```

```
    begin (year_builtP)
```

```
        costruttoreClasseSuper(year_builtP)
```

```
    end
```

```
end module // fine modulo CabooseCar
```

### Osservazioni:

- *r1* è soddisfatto dal costruttore parametrico di RailroadCar, per cui ogni tipo di vagone ereditando la definizione di Railroad deve obbligatoriamente eseguire tale costruttore per avvalorare l'anno di costruzione
- *EngineCar* eredita da una ed una sola classe (esempio di ereditarietà singola)

# Ereditarietà: vagone

## soluzione con ereditarietà: BoxCar

**generic** module BoxCar **inherits**  
RailroadCar, Container;

**public**

**costruttore**(year\_builtP);

**private**

integer height=0, width=0,  
length=0;

function volume(): real;

begin

return height\*width\*length;

end

begin (heightP, widthP, lengthP,  
year\_builtP)

costruttoreClasseSuper(year\_builtP);

height= heightP;

width= width P;

length= lengthP;

end

**end module;**

*/\* year\_built è un attributo di BoxCar  
ereditato dal tipo RailRoadCar \*/*

*/\* percent\_loaded è un attributo di BoxCar  
ereditato dal tipo Container \*/*

*/\* percent\_loaded() è un operatore di  
BoxCar ereditato dal tipo Container \*/*

**Osservazione:** *BoxCar* eredita da due classi (esempio di ereditarietà multipla)

# Ereditarietà: vagone

## soluzione con ereditarietà: TankCar

```
uses Math.PI;
generic module TankCar
inherits RailroadCar, Container;
public
    costruttore(year_builtP);
private
    integer radius=0;
    function volume(): real;
        begin
            return Math.PI*radius*radius;
        end
    begin (heightP, radiusP)
        costruttoreClasseSuper(year_builtP);
        radius= radiusP;
    end
end module;
```

*/\* year\_built è un attributo di TankCar ereditato dal tipo RailRoadCar \*/*  
*/\* percent\_loaded è un attributo di TankCar ereditato dal tipo Container \*/*  
*/\* percent\_loaded() è un operatore di TankCar ereditato dal tipo Container \*/*

**Osservazione:** *TankCar* eredita da due classi (esempio di ereditarietà multipla)

## Esempio vagone: alcune domande

Domanda: Quali vantaggi derivano dall'aver usato l'ereditarietà?

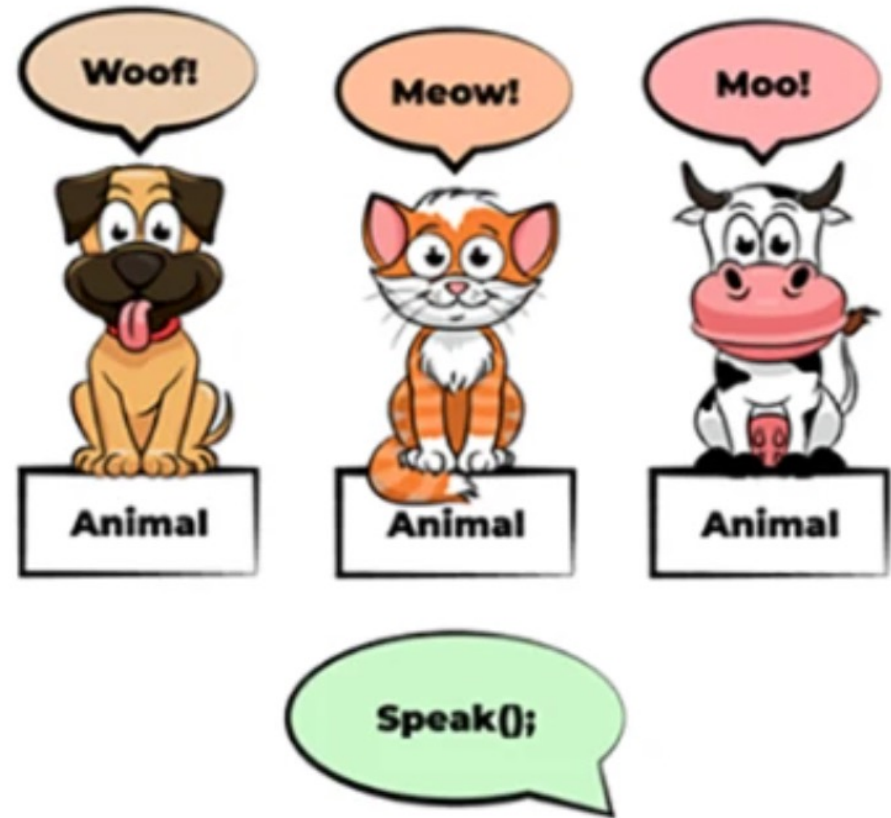
Domanda: Se si volesse aggiungere una proprietà ad EngineCar cosa accadrebbe agli altri tipi di vagoni?

Domanda: cosa accadrebbe se si sbagliasse ad impostare il valore di *default* di *percent\_loaded* nella soluzione senza ereditarietà? Ed in quella con ereditarietà?

Domanda: il requisito che impone che EngineCar e CabooseCar siano entrambi dotati di *volume()* seppur calcolati in modalità differente potrebbe essere usato in fase di modellazione? In questo caso cosa si dovrebbe ereditare?

# Polimorfismo

- Dal greco "pluralità di forme"
  - permette di utilizzare lo stesso *nome* per identificare, ad esempio, operazioni simili che differiscono per la realizzazione
  - Una sola *interfaccia* per molte azioni (*metodi*)
    - ad esempio, in una stessa classe Stack si potrà definire l'operatore **push** tre volte:
      - `push(int element)`
      - `push(char element);`
      - `push(float element);`
  - Il compito di selezionare l'azione specifica da applicare viene delegato al compilatore



## Genericità (Polimorfismo parametrico)...

Possibilità di definire classi parametriche/generiche

- Una classe generica è definita in termini di alcuni "parametri formali"
- La classe viene istanziata solo quando si specificano i parametri attuali corrispondenti ai formali

ESEMPIO: la classe generica *Stack* di elementi di tipo *T*:

**class Stack (T)** (*T* è il parametro formale *generico*)

**NOTA BENE:** una classe generica non è un modulo generico

## ...Genericità (Polimorfismo parametrico)

da cui si potrebbero istanziare Stack contenenti interi o Stack contenenti float:

**Stack (int) sI; // (*int* è il parametro attuale)**

**sI = instantiation of Stack(int);**

**Stack (float) sR; // (*float* è il parametro attuale)**

**sR = instantiation of Stack(float);**

**DOMANDA:** cosa accade in corrispondenza delle 2 istruzioni sottostanti?

**Stack (int) sI;**

**sI instantiation of Stack(real);**

# Dinamicità: polimorfismo verticale

Valutazione a tempo di esecuzione dell'identità degli oggetti (dynamic binding e polimorfismo verticale)

## Magazzino di vagoni su rotaia:

Realizzare un sistema che permetta di avere un magazzino di vagoni su rotaia di vario tipo quali, ad esempio, box car, tank car engine, caboose

### Requisiti

- L'informazione di quali tipi di vagoni saranno inseriti nel magazzino è nota solo al tempo di utilizzo del magazzino ovvero *a tempo di esecuzione*;
- i vagoni sono creati ed inseriti a tempo di esecuzione.

Ed ora proviamo a modellare una soluzione Object-Oriented per gestire un magazzino di vagoni



# Definizione dei vagoni

- Abbiamo già osservato che i diversi tipi di vagoni hanno degli elementi in comune
  - percentuale di carico ricorre sia in Box Car che in Tank Car ed è calcolata nello stesso modo
  - Il volume ricorre sia in Box Car che in Tank Car ed è calcolato in modi differenti
  - anno di costruzione ricorre in tutti i tipi di vagoni

quindi ci serviamo della EREDITARIETÀ per astrarre rappresentazioni e operazioni comuni dei differenti vagoni a rotaia

# Definizione del magazzino

- **Requisiti**

- Un magazzino è un contenitore di differenti tipi di vagoni
- I vagoni possono essere depositati e prelevati dal magazzino

- **Come rappresentare un magazzino?**

- Un magazzino può essere rappresentato come un gruppo di vagoni (elementi/oggetti);
  - Una qualunque struttura dati che rappresenti un gruppo di elementi richiede che tutti gli elementi siano dello stesso tipo (non è il nostro caso dove i vagoni sono di 4, almeno, differenti tipi)

# Definizione del magazzino

In un approccio privo di ereditarietà e polimorfismo potremmo definire un magazzino e rispetto ad esso quattro contenitori, uno per ogni differente tipo di vagone il cui pseudo codice parziale sarebbe

**uses** BoxCar, CabooseCar, TankCar, EngineCar

*module generic* Magazzino

*public*

```
immagazzinaBoxCar(BoxCar); immagazzinaCabooseCar(CabooseCar);  
immagazzinaTankCar(TankCar); immagazzinaEngine(EngineCar);  
prelevaBoxCar(); prelevaCaboose(); prelevaTankCar(); prelevaEngine();
```

*private*

```
var boxes Contenitore BoxCar; cabooses Contenitore CabooseCar; tanks Contenitore  
TankCar; engines Contenitore EngineCar;
```

```
immagazzinaBoxCar(var box: BoxCar);
```

*begin*

```
boxes.aggiungi(box);
```

*end*

*end module*

## Definizione del magazzino

- In un **approccio con ereditarietà**, invece, potremmo fornire una rappresentazione comune a tutti i tipi di vagone su rotaia. Tale rappresentazione è data dalla classe `RailRoad`.
  - L'ereditarietà ci permetterebbe dunque di avere un'unica rappresentazione (contenitore) per tutti i tipi di vagoni
- In un approccio con **ereditarietà e polimorfismo** potremmo considerare tutti gli oggetti dei vari di tipi di vagoni consistenti con il tipo `RailRoad`
  - **Esempio**: un oggetto `Caboose` è anche un oggetto `RailRoad` quindi è possibile trattarlo come l'uno o come l'altro

# Definizione del magazzino: ereditarietà e polimorfismo

*module generic* Magazzino

*public*

*procedure* immagazzinaRailRoad(RailRoad)

*function* prelevaRailRoad(int chiave): RailRoad

*private*

*var* vagoni Contenitore RailRoad // vagoni è una variabile che contiene oggetti

*procedure* immagazzinaRailRoad(RailRoad vagone)

*begin*        vagoni.aggiungi(vagone)

*end*

*function* prelevaRailRoad(int chiave)

*begin*        *restituisce* vagoni.estrai(chiave)

*end*

*costruttore()*

*begin*

vagoni *instantiation of* Contenitore RailRoad

*end*

*end module*

# Creazione dei vagoni

- Le istruzioni per creare un vagone BoxCar in un programma saranno

*uses BoxCar*

*var c1: BoxCar*

*c1 instantiation of BoxCar(2020)*

Da questo momento in poi sarà possibile accedere agli operatori trasparenti dell'oggetto c1 mediante il selettore punto, ad esempio

– *c1.volume()*

- Le istruzioni per creare un vagone Caboose nello stesso programma in cui è stato creato BoxCar saranno

*uses CabooseCar*

*var cab1: CabooseCar*

*cab1 instantiation of CabooseCar(2021)*

Da questo momento in poi sarà possibile accedere agli operatori trasparenti dell'oggetto cab1 mediante il selettore punto, ad esempio

*c1.age()*

# Creazione del magazzino

**uses** BoxCar, CabooseCar, TankCar, EngineCar, Magazzino

**module generic** esempioMagazzino

**public**

costruttore();

**procedure** gestioneMagazzino()

**private**

**var** mag1: Magazzino

costruttore()

**begin** mag1 **instantiation of** Magazzino; **end**

**procedure** gestioneMagazzino()

**begin**

**var** box1,cab1 RailRoad;

box1 **instantiation of** BoxCar(2020);

cab1 **instantiation of** CabooseCar(2021);

mag1.immagazzinaRailRoad(box1);

mag1.immagazzinaRailRoad(cab1);

**end**

**end module**

# Creazione del magazzino: considerazioni

- **module generic** esempioMagazzino
  - ...
  - `mag1.immagazzinaRailRoad(cab1);`
- In un linguaggio di programmazione privo di binding dinamico questa istruzione sarebbe errata a compile-time in quanto c'è inconsistenza di tipo tra
  - `cab1`, che è un oggetto Caboose, ed il tipo di dato accettato in input dal metodo `immagazzinaRailRoad()` che è `RailRoad`

## MA

- Il paradigma Object-Oriented la rende possibile in quanto supporta la consistenza di tipo tra oggetti che sono in relazione di ereditarietà tra loro (combinazione **EREDITARIETÀ** e **POLIMORFISMO**)



# Creazione del magazzino

Qual è il ragionamento?

- *cabl* è un Caboose ma è anche un Railroad quindi a seconda delle necessità potrò considerarlo come l'uno o come l'altro (polimorfismo)

Implicazioni del polimorfismo verticale

- quando *cabl* è usato come Caboose si avranno a disposizione tutti gli operatori definiti per oggetti di tipo Caboose
- quando *cabl* è usato come Railroad si avranno a disposizione tutti gli operatori definiti per oggetti di tipo Railroad

Significato istruzione (slide precedente)

- `mag1.immagazzinaRailRoad(cabl);`
- Significa che *cabl*, pur essendo un oggetto Caboose, è acceduto e manipolato come un oggetto Railroad

Quali sono i vantaggi?

# Creazione del magazzino con binding dinamico

- L'esempio precedente permette di inserire differenti tipi di vagoni a rotaia in un magazzino sapendo a static-time il tipo del vagone da inserire
  - Il tipo della variabile della vagone ed il tipo di oggetto associato ad esso sono noti a static-time
- Cosa accaderebbe se il tipo di vagone da creare, e da immagazzinare poi, fosse noto (solo) a tempo di esecuzione?
  - A static-time non ho conoscenza di quale sia il tipo di vagone da inserire, ho solo conoscenza che sarà un tipo di vagone a rotaia

# Binding dinamico

- A static-time non è noto quale sarà il tipo di vagone che la variabile *vagone* conterrà.
- Solo a tempo di esecuzione alla variabile *vagone* sarà associato un oggetto specifico di un tipo di vagone ferroviario
  - Il tipo dell'oggetto associato a *vagone* è definito a tempo di esecuzione
- Il paradigma Object-Oriented permette questa associazione dinamica a condizione che il tipo della variabile ed il tipo dell'oggetto associato a tempo di esecuzione a tale variabile siano in una relazione di ereditarietà

Come si risolve? Sfruttando ereditarietà, polimorfismo e binding dinamico

## Creazione del magazzino con binding dinamico

```
uses BoxCar, CabooseCar, RailRoad, Magazzino
module generic esempioMagazzino
public
    costruttore()
    procedure immagazzinaRailRoad(vagone: RailRoad)
private
    var mag1: Magazzino. // variabile locale alla classe esempioMagazzino
    procedure immagazzinaRailRoad(vagone: RailRoad)
    begin
        // l'oggetto associato alla variabile vagone, qualunque sia il tipo, è trattato come un oggetto di tipo RailRoad
        mag1.immagazzinaRailRoad(vagone);
    end
    costruttore()
        begin    mag1 instantiation of Magazzino        end
end module
```

# Dinamicità: polimorfismo orizzontale

Valutazione a tempo di esecuzione della realizzazione da invocare:  
variazioni comportamento (polimorfismo orizzontale)

## Esempio:

operatore  $+$  (*somma di numeri oppure concatenazione di stringhe*)

- $3+5$  // *somma di due numeri interi*
- *String prima, seconda;*
- $\text{prima}+\text{seconda}$  // *concatenazione di stringhe*

## Collegamento tra entità:

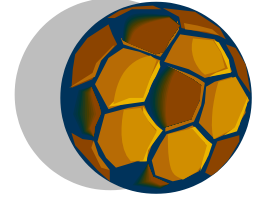
- In un ambiente di programmazione in-the-large, il riferimento fra entità distinte può essere: - risolto prima dell'esecuzione (binding statico) (ad es., MODULA2)
- determinato solo all'esecuzione (binding dinamico) (ad es., SMALLTALK e metodi virtuali in C++)

# Gli oggetti nel paradigma OO

- Gli oggetti incapsulano uno **stato** e un **comportamento**
  - **Stato**: identificato dal contenuto di una certa area di memoria
  - **Comportamento**: definito da una collezione di procedure e funzioni (chiamate **metodi**) che possono operare sulla rappresentazione dell'area di memoria associata all'oggetto.

Da una prospettiva di progetto, gli oggetti modellano le **entità** presenti nel dominio dell'applicazione.

# Gli oggetti: un esempio



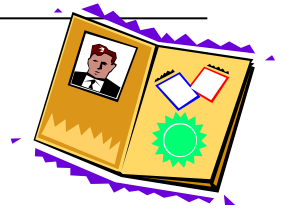
In un gioco elettronico che usa un *pallone*, si può pensare al pallone come un oggetto dotato di uno

- **Stato** espresso in termini di proprietà possedute e dei valori associati in un certo istante di tempo (a tempo di esecuzione):
  - dimensione, posizione in uno spazio di riferimento, etc.
- **Comportamento** espresso in termini di operazioni che può compiere (e che sono accessibili):
  - un pallone può ‘apparire’ o ‘scompare’ dallo schermo, può muoversi, può rimbalzare, etc.

Le variabili ed i metodi definiti nell’oggetto ‘pallone’ stabiliscono, rispettivamente, lo stato e il comportamento che sono rilevanti all’uso del pallone in un gioco elettronico.

Esercizio: descrivere stato e comportamento di un oggetto studente

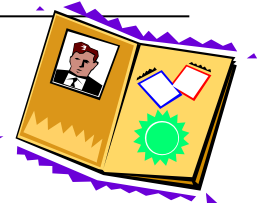
# Identità di un oggetto



- Ogni singolo oggetto ha la propria **identità** ovverosia
- è riconoscibile indipendentemente dal proprio stato corrente
  - ha un **identificatore di oggetto** (*object identifier, OID*) che lo identifica univocamente. In alcuni contesti gli OID sono anche detti **riferimenti** (*references*)
  - Un identificatore di oggetto è **immutabile**
    - non può essere modificato da una qualche opzione di programmazione.
      - di fatto cambiare l’OID di un oggetto equivale alla cancellazione dell’oggetto e alla creazione di un altro oggetto con lo stesso stato.

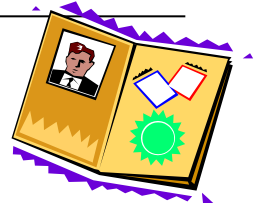


# Identificatore di oggetto



- Assegnazione OID: automatica (solitamente), per cui non hanno un significato nel mondo reale.
  - in molti ambienti di programmazione object-oriented, l'OID corrisponde all'indirizzo dell'area di memoria che conserva lo stato dell'oggetto.
- Accesso agli oggetti: solitamente gli oggetti sono riferiti mediante gli identificatori di variabile
  - Nell'esempio *st1 instantiation of stack10real*;
    - *st1* è l'identificatore della variabile
- Alias: quando variabili distinte possono riferirsi al medesimo oggetto
  - $st2 = st1$

# Identificatore di oggetto: gli Alias



- N.B.: La **presenza di alias** non **significa** che un oggetto non è identificato univocamente, ma semplicemente **che diversi identificatori di variabile sono stati legati al medesimo riferimento di oggetto.**
- **Lo stato di un oggetto può anche contenere il riferimento ad un altro oggetto.** Si dice che un oggetto **punta** ad un altro. Il puntamento è asimmetrico. La simmetria è ottenuta mediante la reciprocità di puntamento.