Object Oriented Programming

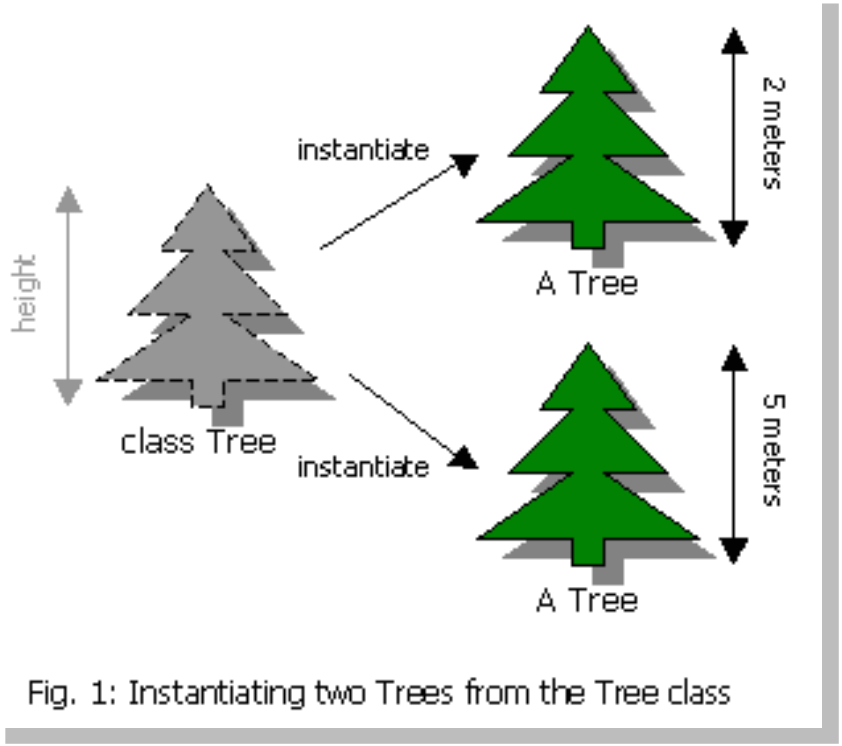


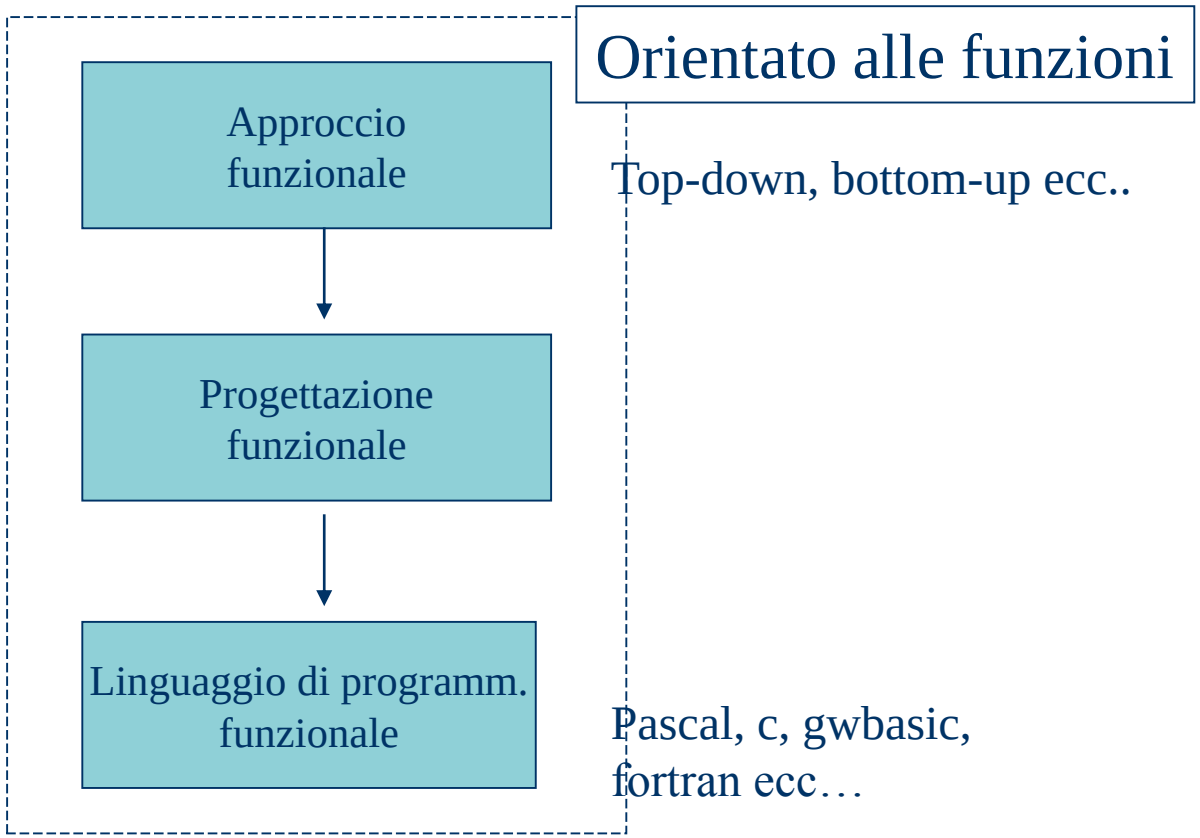
Fig. 1: Instantiating two Trees from the Tree class

Approcci

- Object Oriented è un modo di ragionare. Nel nostro caso si tratta applicarlo al mondo della programmazione.
- Nel mondo del sw commerciale sono da sempre esistiti solo 2 approcci :
  - *Funzionale*
  - *Object Oriented*

Approcci

- Una approccio è la modalità con cui ragiono di fronte ad un problema e il modo con cui penso di trovare una soluzione.



Esempio

Scomposizione funzionale

Diagramma top-down

Creare un videogame

Inizia gioco

Gioca

Salva punteggio

Ferma gioco

Approccio OO

1. Si individuano i protagonisti del problema

1. Si individuano le caratteristiche

1. Si individuano le funzionalità.

…tornando all'Esempio

Approccio Object Oriented

Creare un videogame

Chi è il **protagonista** del problema?

Videogame

caratteristiche

Come è fatto?
-quanti giocatori
-quanti giochi
-…

funzionalità

Cosa posso fare con un videogioco?
- iniziare gioco
- giocare
- salvare punteggio
- …

Classificazione

- Tutti gli oggetti che hanno stesse caratteristiche e stesse funzionalità appartengono ad una stessa **categoria**
- Fare una **classificazione** significa descrivere una categoria di oggetti
- Una **classe** rappresenta una categoria di oggetti

Classe e Oggetto

- Quando si **descrive** un oggetto ci si riferisce sempre alla sua **classificazione**
- Per **usare** un oggetto, ho bisogno di un **rappresentante** della categoria

- **Esempio**:

Non dico "***Questo videogame serve per giocare***",
ma "***I videogame servono per giocare***"

poi giocherò con un **singolo** videogame

non con la categoria di videogame (concetto astratto)

- Un singolo elemento si chiama **OGGETTO** mentre la sua classificazione è la **CLASSE**

Classe e Oggetto

- Un programma OO manipola **oggetti** e agisce unicamente su di essi.

- Le classi sono quindi "***fabbriche di oggetti***" che
  - definiscono l'interfaccia degli oggetti verso il mondo esterno
  - ma ne nascondono l'implementazione
    (incapsulando dati e funzionalità degli oggetti che istanziano)

Classe “VideoGame”

Il protagonista è il videogame

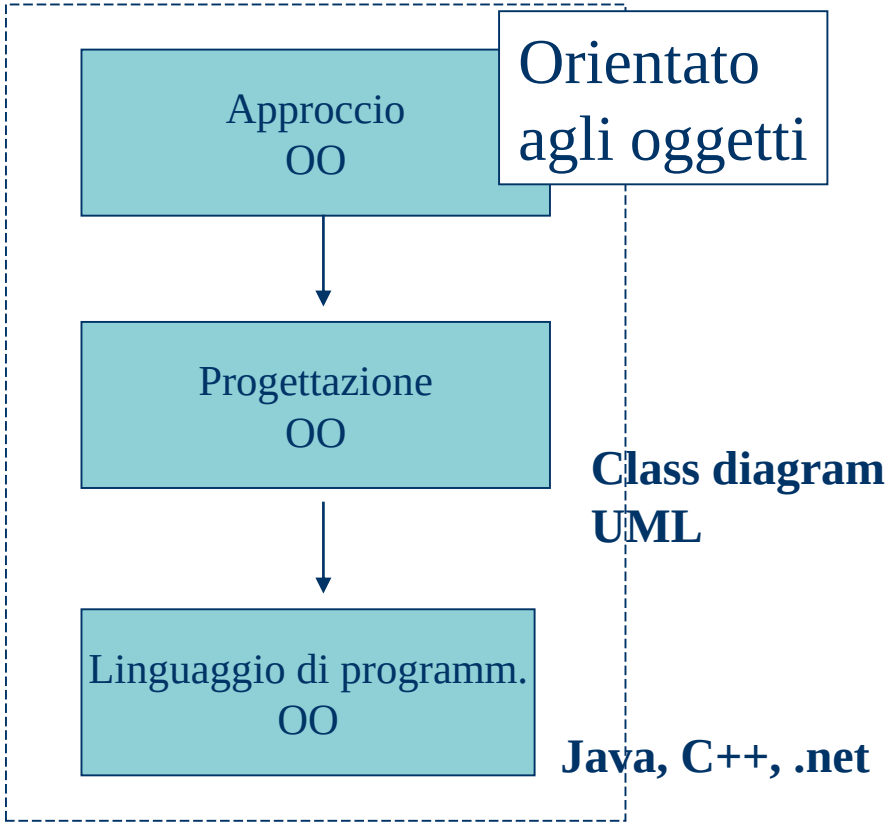
VideoGame
num_giocatori num_giochi marca gioco1 gioco2 …
inizia(qualeGioco) gioca(qualeGioco) salvaPunteggio …

Il *nome della classe* sempre al singolare

Lista di attributi:
componenti di ogni singolo oggetto

Lista di metodi:
funzioni applicabili ad ogni singolo oggetto

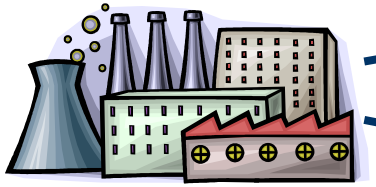
Nuovo approccio



Oggetti

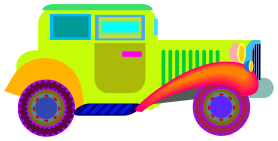
- Tutti gli oggetti di una stessa classe hanno una struttura analoga (attributi) e un comportamento analogo (metodi)

- ***Stesso numero di attributi***
- ***Tutti valorizzati***



Auto
marca modello prezzo colore
accendi spegni cambiaColore

Un'istanza di auto



auto1: Auto
fiat
Punto
10.000 €
verde

Un'istanza di auto

auto2: Auto
citroen
C1
9.000 €
arancione

- Per ottenere oggetti bisogna crearli
- Non è mai necessario invece distruggerli

Istanze

- Ogni oggetto memorizza dati indipendenti che rappresentano il suo **stato**
- Due oggetti (**istanze** di una classe) possono avere gli stessi valori degli attributi ma SONO sempre due oggetti diversi!

Due rose bianche sono due istanze della classe Rosa con lo stesso colore ma non sono la stessa rosa.

Variabili Oggetto

- Per accedere agli oggetti e manipolarli si usano i riferimenti (handler)

Esempio:

Date d;

crea una variabile che può fare riferimento ad oggetti della classe di libreria **Date**.

Variabili Oggetto

IMPORTANTE:

d **non è un oggetto**, ma soltanto una variabile che può **contenere un riferimento** ad un oggetto.

Tornando all'esempio:

Date d;

// d non fa neanche riferimento ad un oggetto,

// Non si possono chiamare metodi della classe Date

Es. **d.setMonth(10);** // non ancora

Variabili Oggetto

Per creare un oggetto occorre:

- l'operatore **new**
- seguito da un metodo speciale detto **costruttore**

Esempio:

```
d = new Date();
// crea un'istanza di Date con la data di sistema
```

Ora d fa riferimento all'oggetto Date appena creato,
ADESSO si possono chiamare i metodi relativi

```
d.setMonth(3);        // setta il mese a Aprile
d.getDay();  // restituisce il giorno della settimana
```

Variabili Oggetto

- Si può dichiarare ed inizializzare una variabile oggetto con un'unica istruzione:

  Date d = new Date();

- Una variabile oggetto può essere esplicitamente impostata a ***null*** <=> non fa riferimento a nessun oggetto

  Date d = null;

- Si possono 2 o più variabili possono puntare allo stesso oggetto.

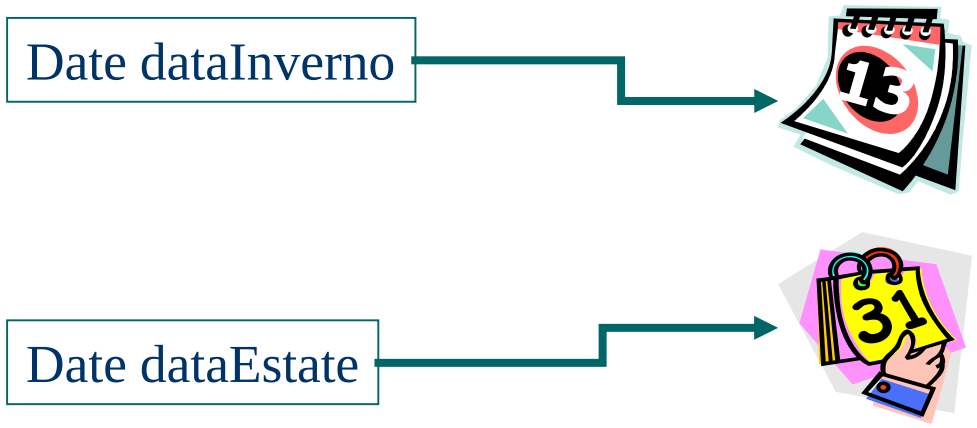
  Se data1 e data2 sono oggetti di tipo Date, si può scrivere

  data1 = data2;

Variabili oggetto

Creo 2 variabili di tipo Date

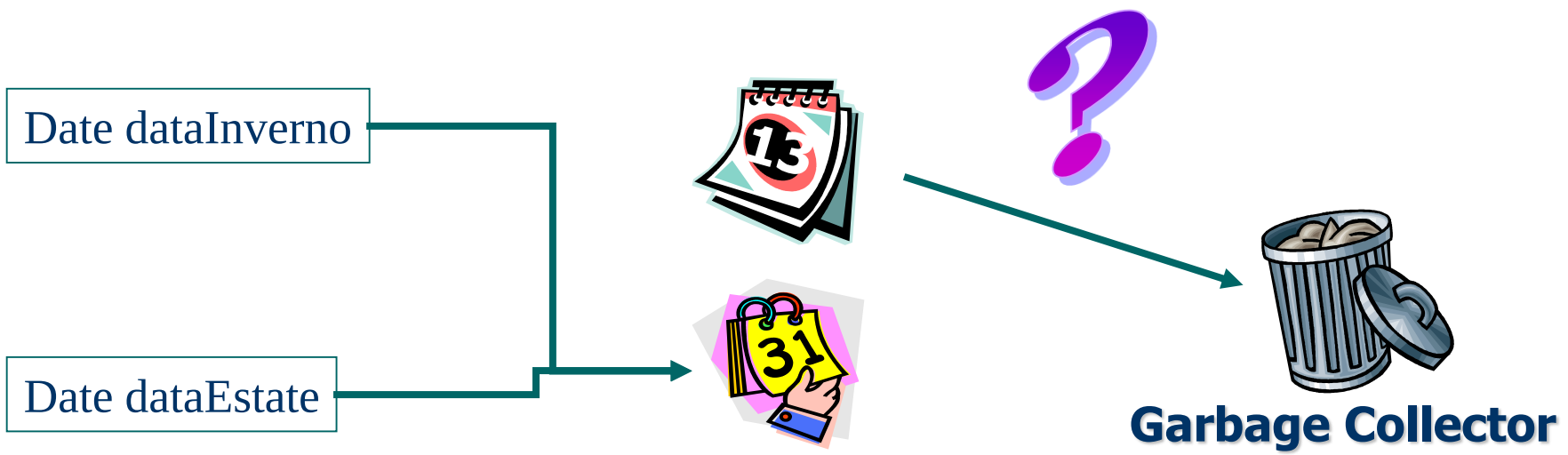
```
// creo il 13 dicembre 2001
Date dataInverno = new Date(2001,12,13);
// creo il 31 luglio 2005
Date dataEstate = new Date(2005,7,31);
```



Variabili Oggetto

Se assegno:

dataInverno = dataEstate;
// entrambi i riferimenti “puntano” a dataEstate
// l’oggetto puntato da dataInverno viene perduto!



Il Garbage Collector

- I tipi primitivi sono allocati direttamente sullo **stack.**
- I tipi oggetto sono allocati dinamicamente per mezzo dell'operatore new. Le variabili riferimento si trovano anch'esse nello **stack**.
- Il **Java heap** è la zona di memoria che contiene gli oggetti

- Il GC è una **speciale routine di sistema** che scandisce il Java Heap liberando la memoria occupata dagli oggetti non più referenziati.

- Il programmatore non può deallocare gli oggetti in modo esplicito
- Invocando **`System.gc()`** si fa partire il GC, ma non si può forzare il suo operato, ne le sue politiche di deallocazione degli oggetti

Passaggio di parametri

- Le variabili **oggetto** sono passate ***"per riferimento"***.
- I **primitivi** sono passati ***"per valore".***

- Al metodo arriva soltanto **una copia** della variabile oggetto
- Il metodo non può modificare il riferimento all'oggetto puntato
- I metodi possono soltanto modificare i contenuti (stato) delle variabili oggetto.

Esercizio: Swap di interi

Definizione di classe

Una classe può essere definita con la seguente forma:

```
public class NomeClasse
{
        [dichiarazione attributi]
        [definizione metodi]
}
```

Si possono anche scrivere 2 o più classi nello stesso nel file, ma solo una classe è **`public`** (quella che da il nome al file).

Le altre sono visibili solo al pacchetto

Definizione di pacchetto

- Un pacchetto è una directory che ha lo scopo di organizzare i file .class che compongono una libreria
- Due aspetti diversi:
  - organizzazione logica dei nomi: ogni package è uno “spazio di nomi” (namespace) distinto
  - organizzazione fisica dei file: ogni package è una cartella distinta del disco
- Il nome di un pacchetto
  - viene esplicitato nella classe (file .java) con l’istruzione

    ```
    package myPackage;
    ```
  - deve essere la prima istruzione della classe
  - il nome deve essere il più possibile univoco

Uso dei pacchetti

- Per usare una classe di un pacchetto si usa l'istruzione

```
import myPackage.MyClass;
import java.util.Date;
```

- Regola di Java
  - tutte le classi devono appartenere ad un package
  - se il programmatore non specifica un package, il compilatore assegna la classe ad un package implicito

- Gli IDE di sviluppo
  - creano le directory di pacchetto a compile time
  - salvano i file .class nelle rispettive directory di pacchetto

Attributi di una classe

- Si dichiarano:

```
tipo1 nomeVariabile1;
tipo2 nomeVariabile2;
```

- I tipi possono essere primitivi o essere Classi di libreria della **Sun**.

  In questo caso bisogna importare il package di appartenenza.

```
import package1.class1;     // importa class1
import package2.*;    // importa tutte le classi di package2
```

- Possono essere richiamati utilizzando la notazione:

```
nomeOggetto.attributo;
```

  Sebbene questo è possibile per le regole di sintassi, è fortemente sconsigliato perché ***indebolisce la robustezza*** della classe.

Metodi di una classe

- Possono ricevere da zero a n parametri di input
- Possono ritornare uno o nessun valore:
  - nel primo caso devono dichiarare il tipo ritornato
  - nel secondo dichiarano ***void***
- Hanno la seguente forma generale:

```
public/private tipoRitornato nomeMetodo(elenco parametriRicevuti)
{
    dichiarazioni di variabili locali;
    istruzioni;
    ……..……;
    [return valore];
}
```

return restituisce il tipoRitornato

- Possono essere richiamati utilizzando la notazione:

```
nomeOggetto.metodo([parametri]);
```

Modificatori

I modificatori descrivono le proprietà di un'entità (classe/metodo/attributo)

relativamente a:

- **visibilità**

  private - protected – public

  definiscono l'insieme di visibilità dell'entità

- **modificabilità**

  final specifica che l'entità non può essere modificata

- **appartenenza** ad una **classe** o alle **istanze**

  static specifica l'appartenenza dell'entità all'intera classe (è condivisa da tutte le istanze)

Uso di final

- **Attributi final**

  - primitivo `final` → valore costante

  - riferimenti ad oggetti `final` → riferimento costante (gli oggetti non possono essere resi costanti)

- **Metodi final**

  impedisce la ridefinizione del metodo nelle sottoclassi (esigenze progettuali)

- **Classi final**

  impedisce di estendere la classe (motivi di progetto o di performance)

Uso di static

- **Attributi statici**

  costituiscono le globali alla classe, condivise da tutti gli oggetti (non sono necessariamente costanti)

- **Metodi statici**

  sono metodi che non necessitano di oggetti per invocarli

  - Non si possono invocare metodi non **`static`** dall'interno di un metodo **`static`**)

  - Non si può utilizzare la parola chiave **`this`**

Costanti

Le costanti pubbliche di classe si dichiarano così:

```
public static final tipo NOMECOSTANTE = value;
```

Sono variabili pubbliche, condivise e non modificabili da tutte le istanze della classe.

La modalità di accesso è come quella degli attributi:

```
nomeOggetto.costante
```

Classi Interne

- E' possibile definire una classe all'interno di un'altra classe.
- Usare classi interne consente:
  - di raggruppare classi dal punto di vista logico
  - controllarne la visibilità dalla classe esterna

    (l'accesso è subordinato ai metodi e al nome della classe esterna)
- Rappresentano una composizione logica

Enumeration

- E' un tipo di dato speciale i cui valori appartengono ad un insieme finito di elementi.
- Il tipo **`enum`** è typesafe, cioè garantisce il range.

- Per definire una **`enum`** è necessario indicare tutti i suoi valori.
- L'accesso ai valori si ottiene con **`nomeEnum.valore`**

Esempio:

```
enum ValoriCarteGioco {due, tre, quattro, cinque, sei, sette,
                       otto, nove, dieci, jack, donna, re, asso};
enum SemiCarteGioco {fiori, quadri, cuori, picche};
```

Metodi statici

- Ogni **`enum`** possiede alcuni metodi statici:
  - **`NomeEnum valueOf(String nome)`**

    ritorna la costante **`enum`** data dalla stringa **`nome`**
  - **`NomeEnum[] values()`**

    ritorna un array con tutte le costanti della **`enum`** nell'ordine in cui sono state definite
  - **`int compareTo(Enum enum)`**

    ritorna un numero positivo/negativo/zero a seconda che il valore enum corrente sia maggiore/minore/uguale di quello parametro. L'ordine è quello indotto dalla dichiarazione dell'enum

Esempio di utilizzo

Classe Carta

```
public class CartaGioco {
    private final ValoriCarteGioco valore;
    private final SemiCarteGioco   seme;

    public CartaGioco (ValoriCarteGioco v, SemiCarteGioco s) {
          this.valore = v;
          this.seme = s;
    }
}
```

main()

```
public class Gioco {
    public static void main(String[] args) {
          ValoriCarteGioco value = ValoriCarteGioco.asso;
          SemiCarteGioco seed = SemiCarteGioco.fiori;
          CartaGioco card1 = new CartaGioco(value,seed); // OK
          CartaGioco card2 = new CartaGioco(seed,value); // Non compila
    }
}
```

Esercizi

- Facciamo qualche esercizio con le classi di libreria!