OOP

Object Oriented Programming
Parte 1

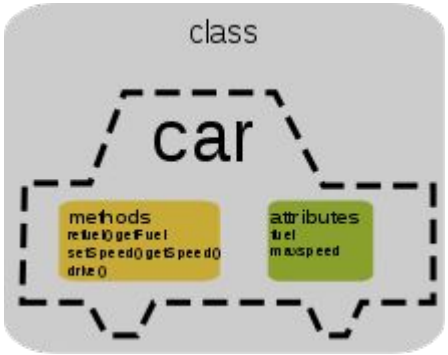
MODELLI INFORMATICI

Definizione: Un modello astratto è la rappresentazione formale di idee e conoscenze relative a un fenomeno o a un'entità.

La stessa realtà può utilmente essere modellata in modi diversi, e a diversi livelli di astrazione.

Classe

La **class** è il costrutto utilizzato in un linguaggio object oriented per definire e modellare un'entità software.



Una classe contiene:

- **informazioni**
- **comportamento**

Le **informazioni** sono memorizzate per mezzo degli **attributi**: variabili dichiarate all'interno della classe.

Il **comportamento** è dato attraverso i **metodi**: funzioni programmate all'interno della classe.

Classe e oggetti

La classe rappresenta il **concetto**, **l'idea** di come una certa entità è pensata all'interno del software.

Una volta definita la classe è possibile creare infinite sue **istanze**, cioè **oggetti concreti**, ognuno dei quali rispetta la programmazione della classe a cui appartiene ed è caratterizzato dalle proprie informazioni che lo contraddistinguono.

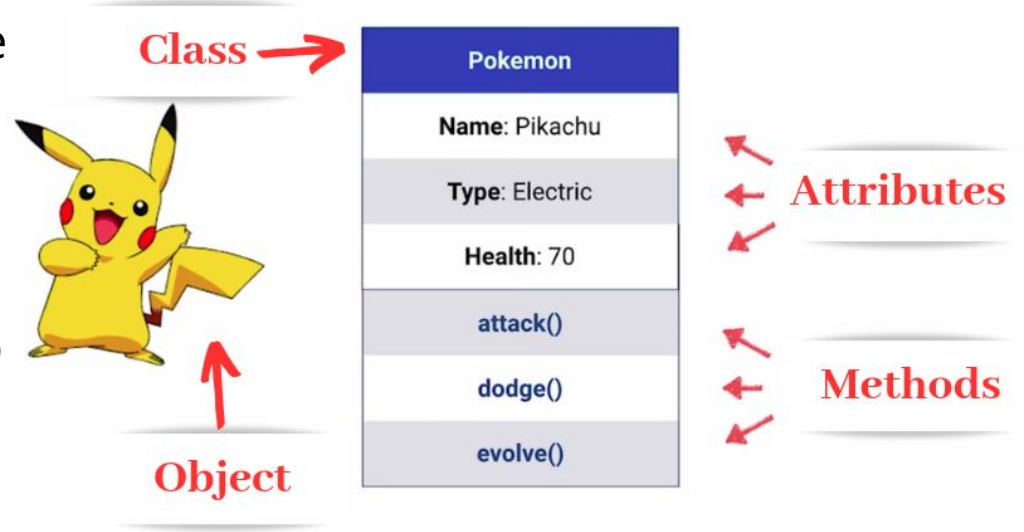
Attenzione: classe e oggetto NON sono sinonimi.

Classe e oggetto

Tutti i Pokemon possiedono certe capacità (metodi) e caratteristiche (attributi).

Pikachu, in quanto pokemon, ha un nome, una tipologia e un valore di salute ben definiti che lo caratterizzano.

Inoltre sà attaccare, schivare ed evolversi.



Class VS Struct

Anche le struct possono rappresentare entità software.

Anche le struct possono contenere dati e funzioni.

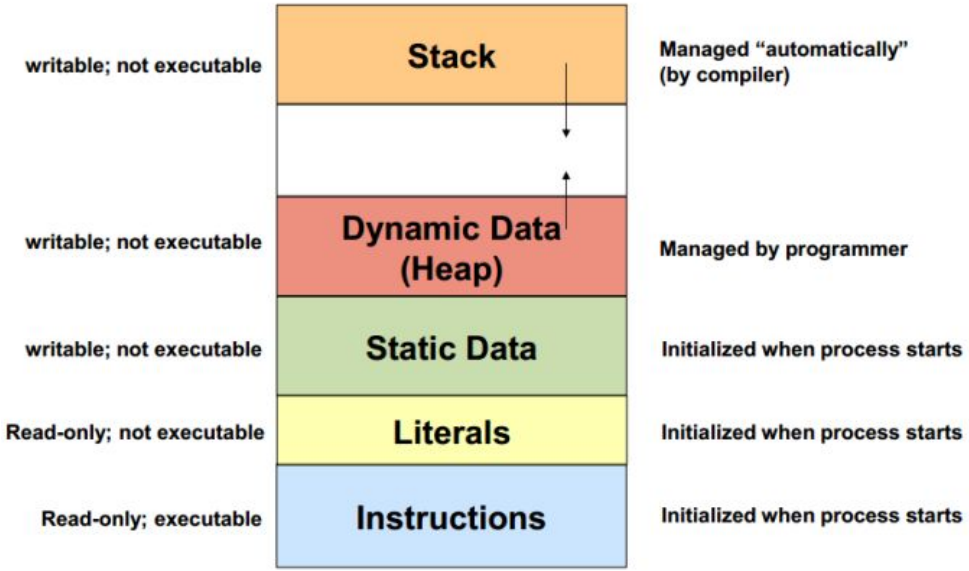
Perché abbiamo bisogno delle classi e degli oggetti?

Organizzazione della memoria

Stack e **Heap** sono due zone di memoria che il sistema assegna ad un programma in esecuzione.

Stack: zona di memoria statica (fissa) perché determinata dal compilatore e non estendibile al bisogno.

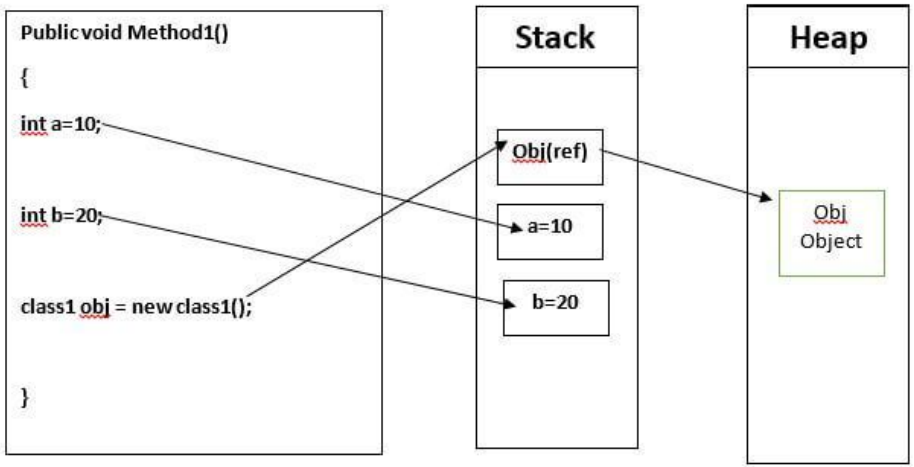
Heap: zona di memoria dinamica, a disposizione del programmatore. Se il programma necessita di memorizzare grandi quantità di dati, il sistema incrementa l'heap.



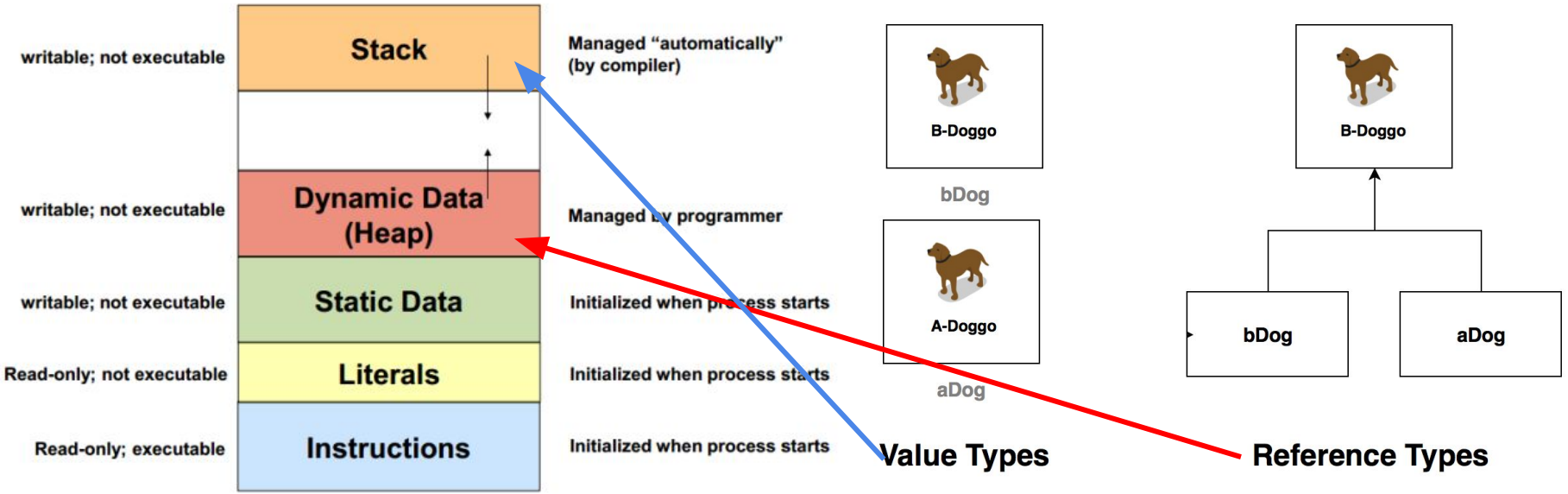
Organizzazione della memoria

Lo **Stack** memorizza le variabili che hanno il tipo di valore (es. int, double, **struct**, ecc...)

L'**Heap** memorizza le variabili che sono referenziate da un puntatore (es. quando usiamo ref o out, gli **oggetti**)



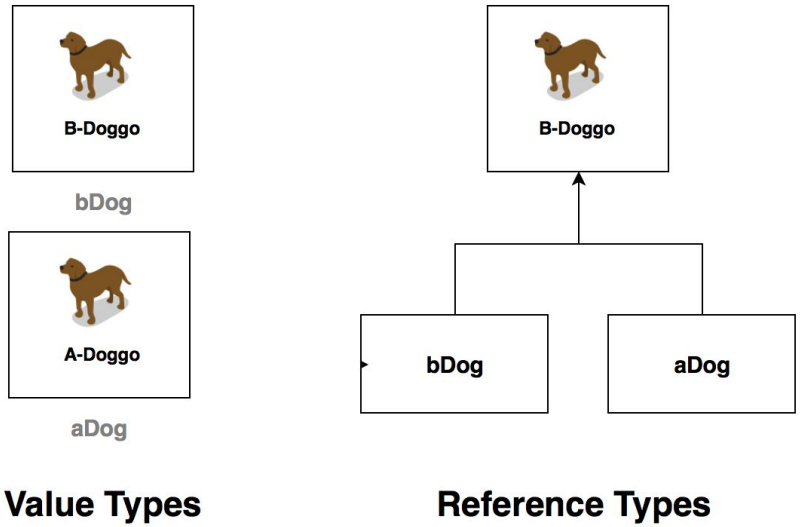
Class VS Struct a livello tecnico



Class VS Struct a livello operativo

La **struct** è un tipo di **valore**: questo significa che le variabili di tipo struct sono sempre delle copie.

La **class** è un tipo di **riferimento**: questo significa che ogni variabile è il puntamento all'unica cella di memoria che contiene il dato e non una copia del dato.



Quando usare le struct?

Usare le **struct** quando:

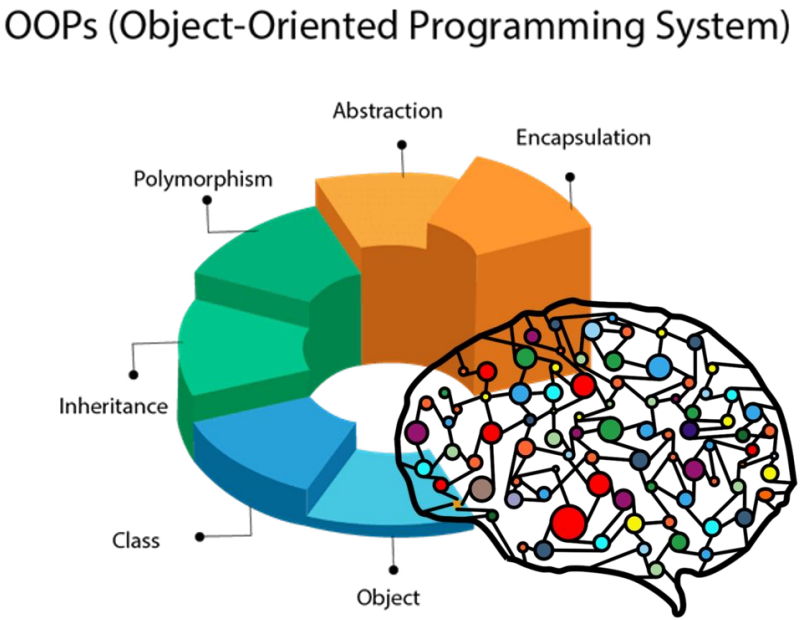
- contengono **pochi dati** e di tipo primitivo
- il loro ciclo di **vita è breve**
- non devono essere **manipolate**/modificate o passate a funzioni
- si ha la necessità di crearne **tante** (la creazione di una struct è molto più veloce rispetto la creazione di un oggetto)

Lo Stack è di piccole dimensioni e fisso, quindi se le variabili struct sono piccole e vengono distrutte nel breve termine allora OK 👍

Quando usare gli oggetti?

Usare gli **oggetti** quando:

- **sopravvivono** per molto tempo o per tutta la durata del programma
- contengono **molti dati** e/o altri oggetti complessi
- si vogliono sfruttare tecniche avanzate come **ereditarietà** e/o **polimorfismo**



L'Heap è dinamico e grande quindi non ci sono problemi nel memorizzare grandi variabili e/o numerose per molto tempo. 👍

Accessibilità e information hiding

Organizzazione del codice

Per meglio organizzare il codice si utilizzano tecniche di confinamento delle varie sottoparti che compongono un software.

Abbiamo già visto una prima tecnica: il **namespace**

Ora introduciamo il concetto di accessibilità.

Accessibilità

Tecnica che **regola** l'utilizzo di elementi del codice come variabili o funzioni.

L'accessibilità può **impedire** l'utilizzo di un elemento (caso delle funzioni) o **limitare** solamente l'utilizzo in lettura o in scrittura (caso delle variabili).

Oltre che ad essere una tecnica di organizzazione del codice, all'interno dell'**OOP** il concetto di accessibilità rientra nel principio dell'**information hiding**.

Information hiding

Principio secondo il quale l'implementazione interna di un componente software debba essere **protetta**, e quindi nascosta, in modo che non sia possibile modificarla dall'esterno.

In tale contesto è essenziale poter regolare **l'accesso** alle parti software per attuare tale principio.

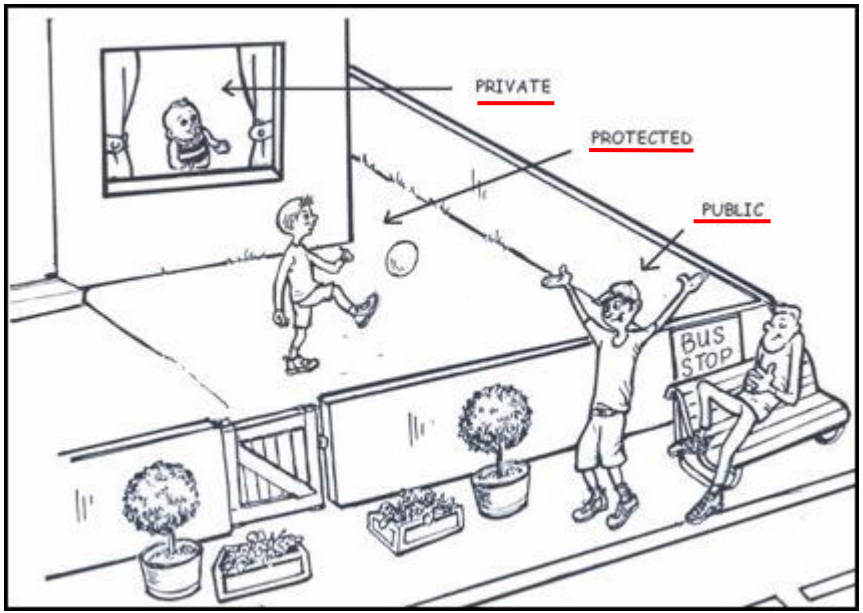
Livelli di accessibilità

public: il tipo o il membro può accedere a **qualsiasi** altro codice nello stesso assembly o in un altro assembly che vi fa riferimento.

private: il tipo o il membro è accessibile solo dal codice nella **stessa** class o struct .

protected: il tipo o il membro è accessibile solo dal codice nello **stessa** class o in una class **derivata** da tale class .

internal: il tipo o il membro è accessibile da qualsiasi codice nello **stesso assembly**, ma non da un altro assembly.



***assembly: programma o libreria**

Livelli di accessibilità - defaults

Accessibilità di default: livello di accessibilità applicato in automatico se il programmatore non esprime preferenza.

In generale viene applicato il **massimo livello di protezione disponibile** per l'elemento in oggetto.

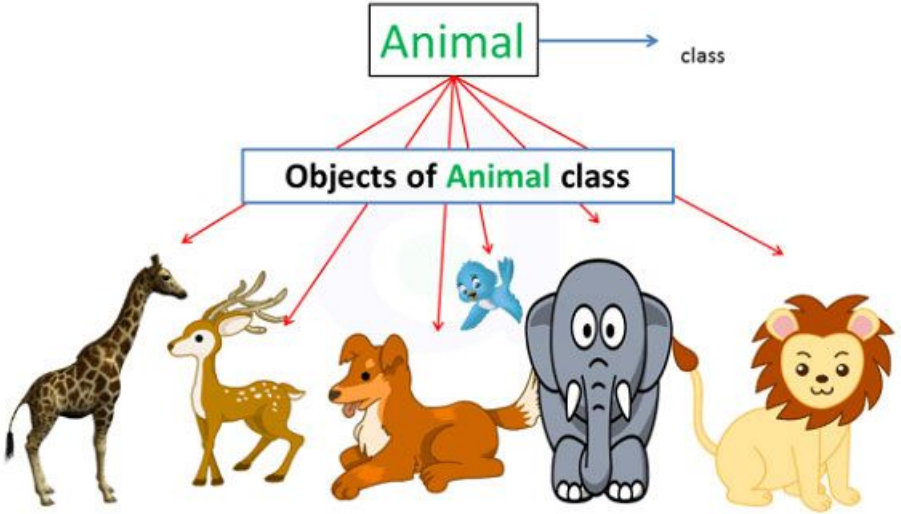
Class, struct e interface: internal
Membri di class o struct: private
Membri di interface: public

Tecniche dell'OOP

Ereditarietà

Nel OOP si possono creare entità derivanti da altre entità più astratte di livello superiore.

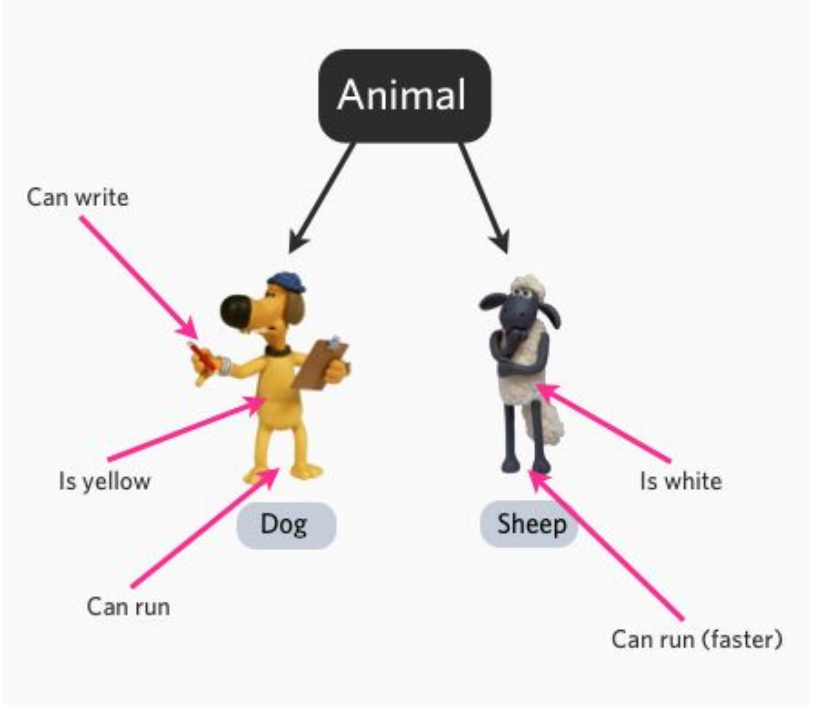
Il **Cane** è un'entità concreta e dettagliata, "figlia" dell'entità più astratta **Mammifero**, la quale si può ricondurre all'entità ancora più astratta **Animale**.



Ereditarietà

Ogni classe che rientra in una catena di ereditarietà **acquisisce** dalla classe "padre", o superiore, quei metodi e attributi marcati come **public** o **protected**.

Ogni classe "figlia" (o concreta) può anche definire nuovi metodi e nuovi attributi che appartengono solamente a lei.



Esempio di una classe

```
public class Somma
{
    public double Operando1 { get; private set; }
    public double Operando2 { get; private set; }
    public double Risultato { get; private set; }

    public Somma(double n1, double n2)
    {
        Operando1 = n1;
        Operando2 = n2;
    }

    public void Esegui()
    {
        Risultato = Operando1 + Operando2;
    }

    public override string ToString()
    {
        return String.Format("{0} + {1} = {2}", Operando1, Operando2, Risultato);
    }
}
```

```
Somma s1 = new Somma(4, 7);
s1.Esegui();
Console.WriteLine(s1); // 4 + 7 = 11

Somma s2 = new Somma(8, s1.Risultato);
s2.Esegui();
Console.WriteLine(s2); // 8 + 11 = 19
```

Incapsulamento

L'implementazione dei due operandi viene nascosta all'esterno.

I due operandi vengono valorizzati grazie all'incapsulamento svolto nel metodo costruttore.

```
public class Somma
{
    private double operando1;
    private double operando2;
    public double Risultato { get; private set; }

    public Somma(double n1, double n2)
    {
        operando1 = n1;
        operando2 = n2;
    }

    public void Esegui()
    {
        Risultato = Operando1 + Operando2;
    }

    public override string ToString()
    {
        return String.Format("{0} + {1} = {2}", Operando1, Operando2, Risultato);
    }
}
```

incapsulamento

passaggio di parametri

```
Somma s1 = new Somma(4, 7);
s1.Esegui();
Console.WriteLine(s1); // 4 + 7 = 11

Somma s2 = new Somma(8, s1.Risultato);
s2.Esegui();
Console.WriteLine(s2); // 8 + 11 = 19
```

Perché l’incapsulamento?

L'uso più frequente dell'incapsulamento è quello di **nascondere** lo strato fisico in cui vengono memorizzati i dati; in tal modo, se la rappresentazione interna dei dati cambia, le modifiche si propagano soltanto ad una piccola parte del programma.

Supponiamo, ad esempio, che un punto di uno spazio tridimensionale sia rappresentato dalle coordinate x,y,z espresse da tre valori scalari in virgola mobile, e che, successivamente, si passi ad una rappresentazione delle coordinate del punto mediante un singolo array a tre dimensioni: un modulo di programma progettato secondo la tecnica dell'incapsulamento **proteggerà** il resto del programma da questo cambiamento di rappresentazione.