Ereditarietà

Programmazione Avanzata e Parallela
2022/2023

Alberto Casagrande

Classi e Gerarichie di Nozioni

In taluni casi, le classi specificano in dettaglio nozioni già implementate

Es. `Animal` e `Cat`, `Device` e `Mobile` o `Human` e `Student`

Le classi già implementate potrebbero fornire
delle funzionalità che non vorremmo re-implementare

Es. `Animal::num_of_legs()`, `Device::is_electronic()`, e `Human::name()`

Ereditarietà

Definisce un sottotipo del tipo descritto da una classe

La sottoclasse **eredita** i membri e i metodi della superclasse

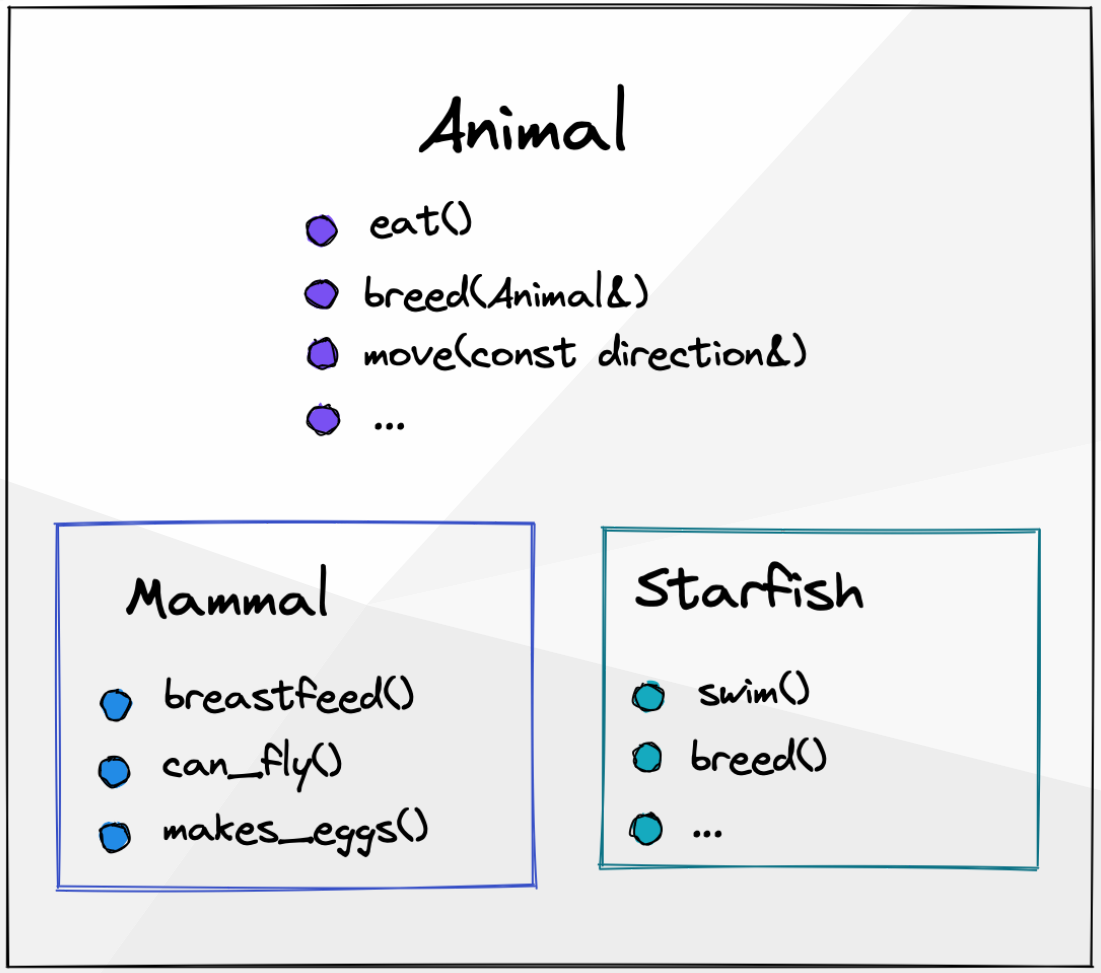
L'**ereditarietà** definisce una **sottoclasse** di una classe (**superclasse**)

La sottoclasse è anche detta **classe derivata**

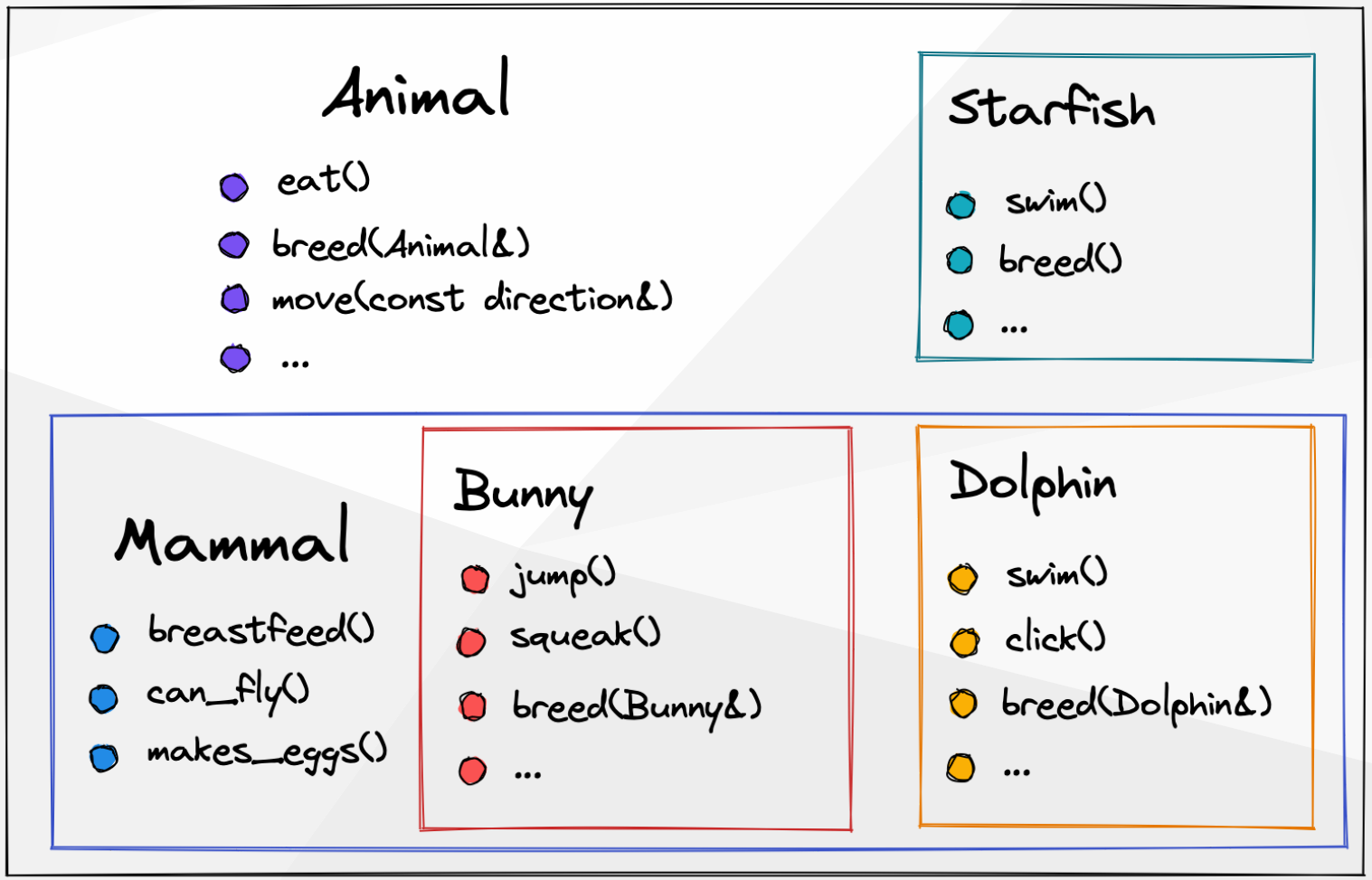
Il DAG di derivazione di una classe è detto **gerarichia della classe**

`Animal` e ...

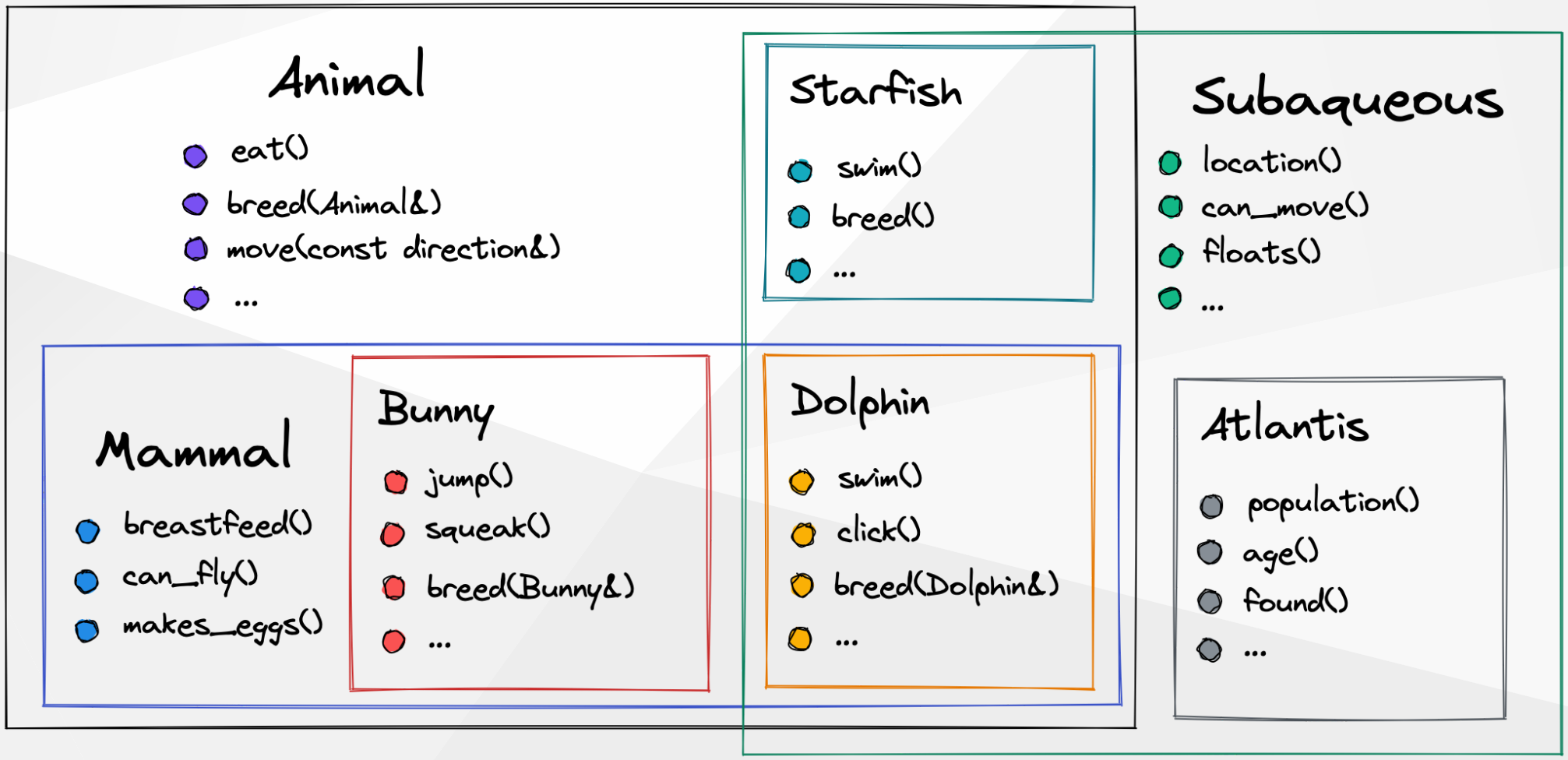
`Animal` e Le Sue Derivate



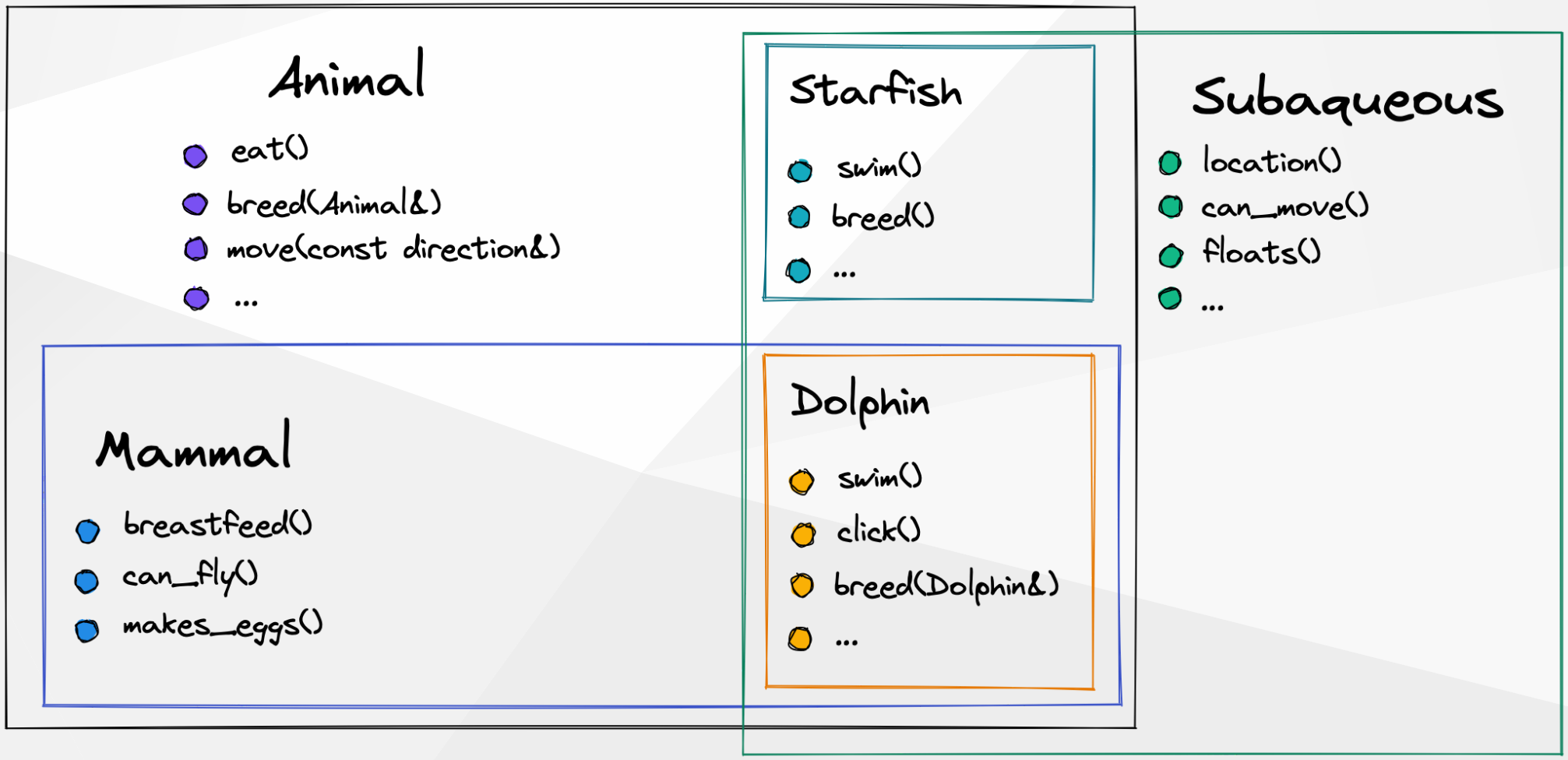
`Animal`, Le Sue Derivate e Le Loro Derivate



Ereditarietà Multipla



Ereditarietà Multipla



Classi Derivate e Metodi

Le classi derivate ereditano tutti i metodi delle superclassi

```
Bunny b;

b.can_fly();  // OK: can_fly() è un metodo di Mammal, superclasse di Bunny
b.eat();      // OK: eat() è un metodo di Animal superclasse di Mammal

b.swim();     // ERRORE: swim() non è un metodo nella gerarchia di Bunny
```

```
Dolphin d;

d.eat();      // OK: eat() è un metodo di Animal
d.floats();   // OK: floats() è un metodo di Subaqueous
```

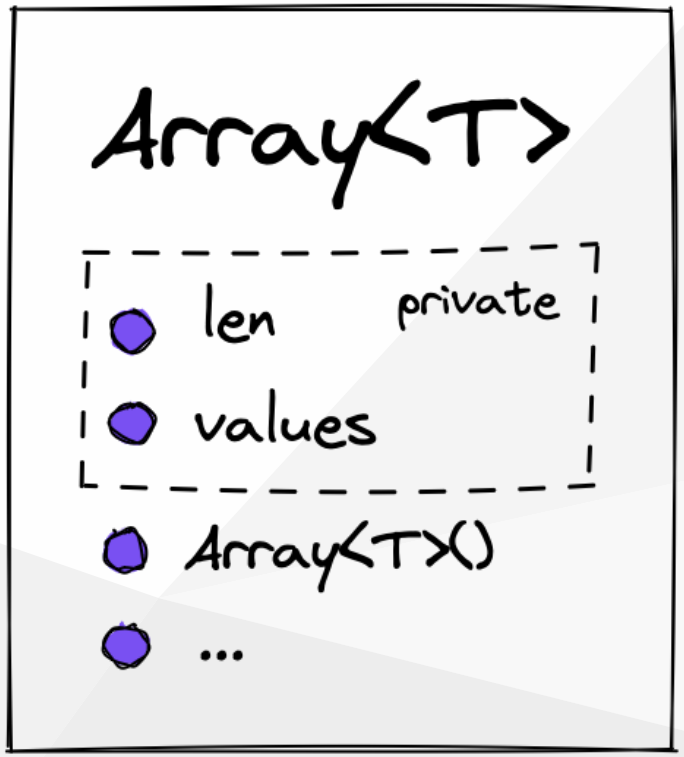
Modalità di Accesso

In C++ alcuni membri/metodi di una classe sono privati

Essi non sono accessibili a nessun metodo/funzione che non sia implementata nella classe stessa

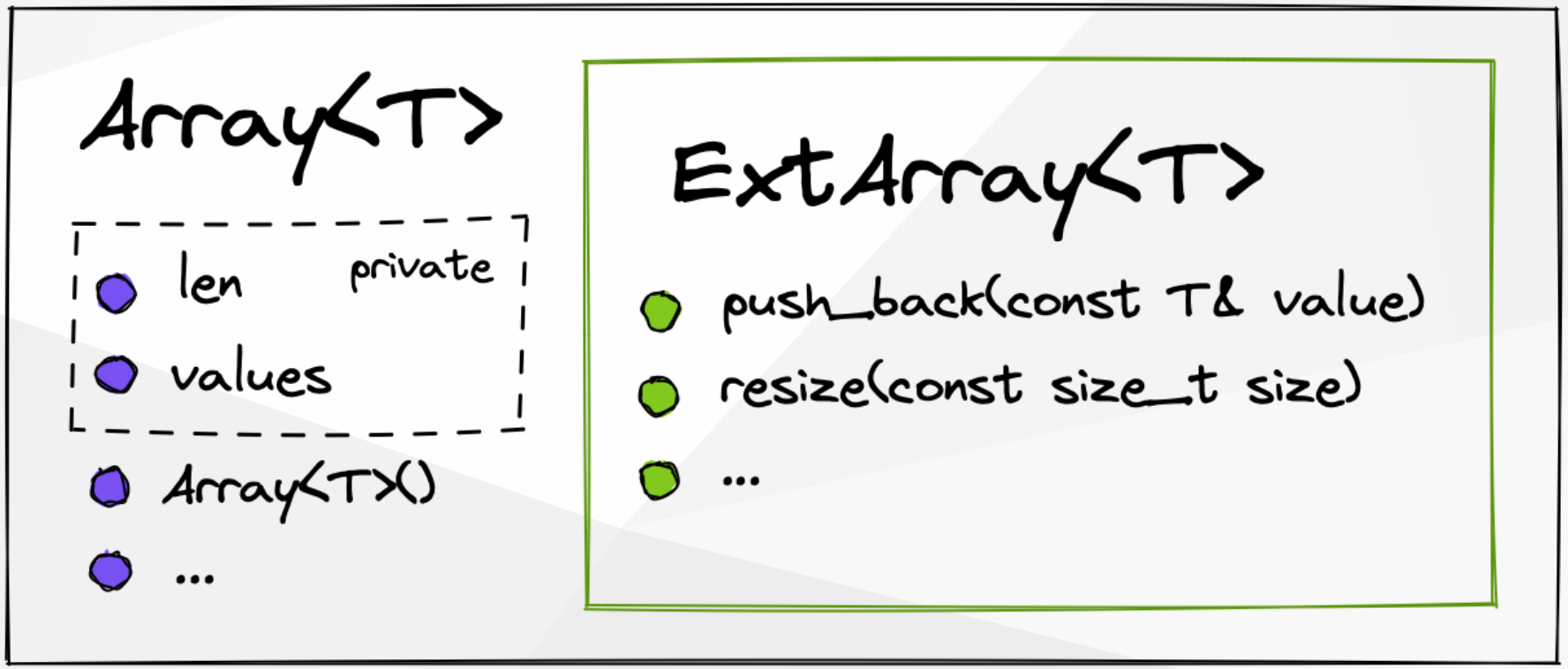
Non sono accessibili nemmeno alle sottoclassi

La Classe `Array<T>`



Entrambi i membri di `Array<T>` sono privati

Supponiamo di Estendere `Array<T>`



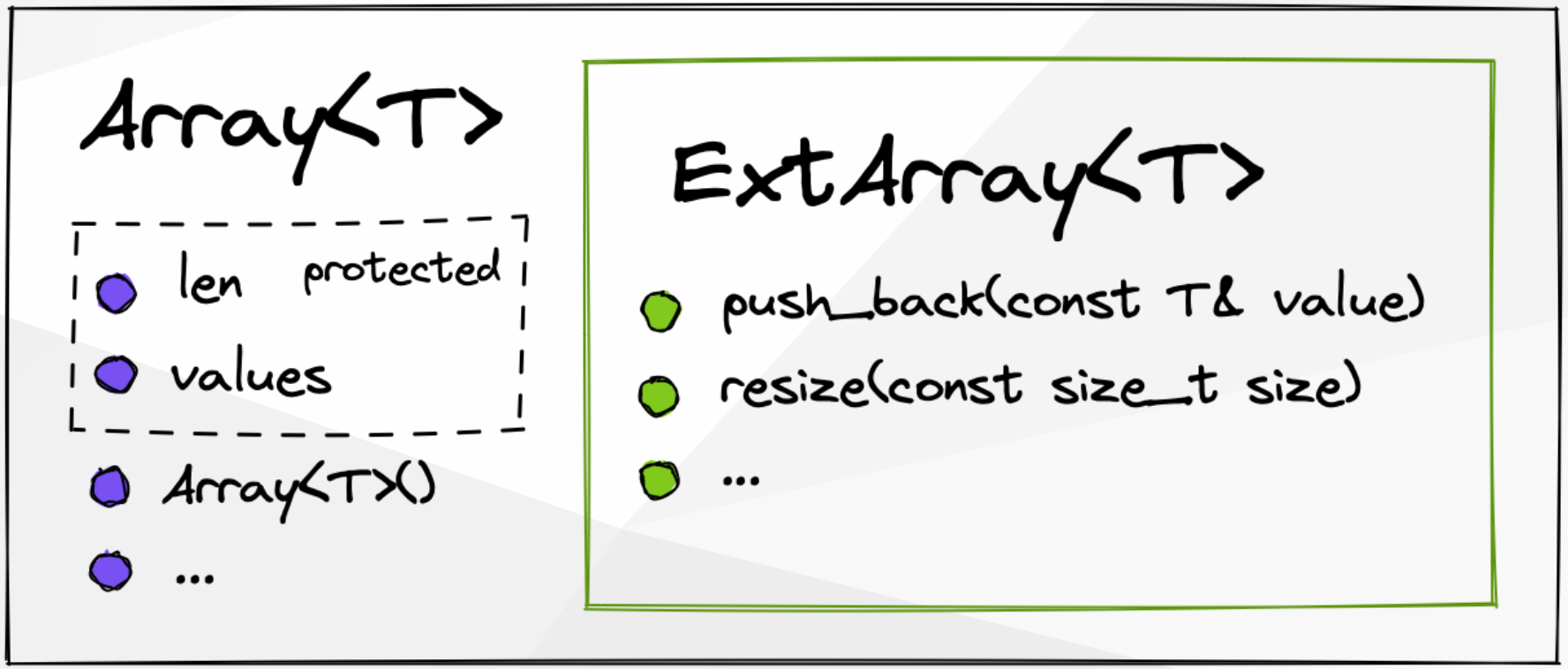
`ExtArray<T>` non può accedere ai membri privati di `Array<T>`

Lo Specificatore d'Accesso `protected`

I membri/metodi

- `public` sono accessibili da chiunque
- `private` sono accessibili solo dai metodi della classe
- `protected` sono accessibili accessibili dai metodi della *classe* e delle sue *derivate dirette*

Membri `protected` per `Array<T>`



Ora i metodi di `ExtArray<T>` possono accedere ai membri di `Array<T>`

Accesso ai Membri/Metodi Ereditati

In alcuni casi, potremmo voler limitare l'accesso alla superclasse

Potremmo voler rendere privati i metodi pubblici della superclasse

Quando deriviamo una classe specifichiamo il tipo di accesso

Tipo	Membri Publici	Membri Protetti	Membri Privati
`public`	pubblici	protetti	non accessibili
`protected`	protetti	protetti	non accessibili
`private`	non accessibili	non accessibili	non accessibili

Classe Derivata in C++

```
class A {
    int m1;
  protected:
    int m2;
  public:
    int m3;
}

class B: public A {
  // m1 è privato in A e non è accessibile in B
  // m2 è protetto in A e privato in B
  // m3 è pubblico
}
```

Ereditarietà e *Overriding*

L'**overriding** consente di "re-implementare" i metodi nelle derivate

```
struct A {
    void print() {
      std::cout << "A::print()" << std::endl;
    }
}

struct B : public A {
    void print() {
      std::cout << "B::print()" << std::endl;
    }
}
```

Calcolare l'area di una `Shape`

Consideriamo le classi `Square` e `Triangle` uniche derivate da `Shape`

Queste implementano `Square::area()` e `Triangle::area()`

Perciò di tutti gli `Shape` sappiamo calcolare l'area

Come calcolare l'area di un oggetto puntato da `Shape*`?

```
Shape* s = new Square(3);

s->area(); // come fare?
```

Distruttori e Classi Derivate

Cosa succede se de-allochiamo uno `Square` da `Shape*`?

```
struct Shape {
  ~Shape();
};

struct Square: public Shape {
  ~Square();
};
...

Shape* s = new Square(3);

delete s;  // Non viene invocato il distruttore di Square!!!!!
```

Metodi Virtuali

Consentono l'overriding su riferimenti della superclasse

```
struct Shape {
  virtual double area() {...}
};

struct Square: public Shape {
  double area() {...}
};
...

Shape* s = new Square(3);

s->area(); // viene invocato Square::area()
```

Distruttori Virtuali

Anche i distruttori possono essere resi virtuali

```
struct Shape {
  ~Shape() { std::cout << "Deleting Shape" << std::endl; }
};

struct Square: public Shape {
  ~Square() { std::cout << "Deleting Square" << std::endl; }
};
...

Shape* s = new Square(3);

delete s;  // Viene invocato il distruttore di Square
```

Metodi Virtuali e `override`

`override` può etichettare i metodi che sovrascrivono metodi virtuali

```
struct Shape {
  virtual double area();
  bool in_2D();
};

struct Square: public Shape  {
  double area() override;

  bool in_2D() override;        // ERRORE: Shape::in_2D() non è virtuale

  bool is_fractal() override;  // ERRORE: Shape::is_fractal() non esiste
};
```

Metodi Virtuali Puri

Come calcolare l'area di una `Shape`?

```
struct Shape {
  virtual double area() { ??? }
};
```

Non ha senso implementare il metodo `Shape::area()`

```
struct Shape {
  virtual double area() = 0;  // questo è un metodo puro: dovrà essere
                              // implementato dalle classi derivate
};
```

Metodi Virtual Finali

Non si può fare l'*override* di metodi `final`

```
struct Shape {
  bool is_a_shape() final;
};

struct Square: public Shape  {
  bool is_a_shape();  // ERRORE: Shape::is_a_shape() è finale
};
```