

Classe

Una classe è un modello / stampo, definisce cosa gli oggetti avranno (campi, metodi, costruttori).

Di per sé, una classe è solo definizione → non occupa memoria per i dati (tranne campi static, che appartengono alla classe).

Esempio:

```
class Cane {  
    string nome;  
    int eta;  
  
    void abbaia() {  
        system.out.println(nome + "sta abbaiando!");  
    }  
}
```

Qui la classe **Cane** è solo la descrizione di come deve essere un cane: avrà un nome, un'età e potrà abbaiare

Significato di public, private e protected

Servono per il controllo di accesso (visibilità dei membri della classe):

- **public** → accessibile ovunque.
- **private** → accessibile solo dentro la stessa classe.
- **protected** → accessibile nella classe, nello stesso package e nelle sue sottoclassi.

In java: se non scrivi nulla è package-private (visibile solo nello stesso package)

In C++: se non scrivi nulla è private

Ricorda: public < protected < package-private < private

Visibilità classi

	Classe in cui è dichiarata	Sottoclassi	All'esterno della classe
public	✓	✓	✓
private	✓	✗	✗
protected	✓	✓	✗

Oggetto (= istanza)

un oggetto è un'istanza nella classe.

Quando fai new, stai costruendo in memoria una copia concreta di quella "ricetta".

Ogni oggetto ha i suoi valori personali dei campi.

Esempio:

```
public class Main {  
    public static void main(String[] args) {  
        Cane c1 = new Cane();  
        c1.nome = "Jenny";  
        c1.eta = 10;  
  
        Cane c2 = new Cane();  
        c2.nome = "Sam";  
        c2.eta = 3;  
  
        c1.abbaia(); // "Jenny sta abbaiano!"  
        c2.abbaia(); // "Sam sta abbaiano!"  
    }  
}
```

La classe è lo stampo.

c1 e c2 sono oggetti distinti, ognuno con i propri dati (nome, età)

Variabile

Variabile è un termine generale → è ogni contenitore di un valore in un programma

In Java può essere:

1. Variabile locale (dentro a un metodo):

```
void saluta() {  
    string msg = "Ciao!"; // variabile locale  
}
```

2. Parametro di metodo:

```
void setNome( string n ) { // parametro n  
    nome = n;  
}
```

3. Campo:

```
Class Persona {  
    // campi = attributi della classe / oggetto  
    String nome; // ogni oggetto avrà il suo nome  
    int eta; // ogni oggetto avrà la sua età  
}  
  
Public class Main {  
    public static void main(String[] args) {  
        Persona p1 = new Persona();  
        p1.nome = "Anna";  
        p1.eta = 20;  
  
        system.out.println(p1.nome + " ha " + p1.eta + " anni");  
    }  
}
```

Tipo	Java	C++
Int	Int a = 10;	Int a = 10;
Float	Float b = 3.14f;	Float b = 3.14;
Double	Double c = 3.14159;	Double c = 3.14159;

Char	<code>Char d = 'A';</code>	<code>Char d = 'A';</code>
String	<code>String e = "ciao";</code>	<code>String e = "ciao";</code>
Array	<code>Int[] arr = new int[] {1, 2, 3};</code> <code>Int[] arr = new int[];</code>	<code>Int arr[] = {1, 2, 3};</code>

Costruttore

È una funzione speciale dentro la classe che serve a creare e ad inizializzare un nuovo oggetto.

Ha lo stesso nome della classe e NON ha tipo di ritorno.

In Java se non lo scrivi, il compilatore ne crea uno “vuoto” (costruttore di default)

Esempio:

```
class Persona {
    string nome;
    int eta;

    Persona(string n, int e) {
        Nome = n;
        eta = e;
    }
}

Public class Main {
    Public static void main (string[] args) {
        Persona p = new Persona("Anna", 23);
        // costruttore chiamato qui
        system.out.println(p.nome); // "Anna"
    }
}
```

Differenza con metodo normale: si chiama automaticamente con new, non ha void o tipo di ritorno

Metodo

È una funzione dentro la classe.

Serve a descrivere un comportamento (cosa un oggetto sa fare).

Può avere tipo di ritorno (int, string, void, ...), parametri di visibilità (public, private, ...).

Esempio:

```
class Persona {  
    string nome;  
    int eta;  
  
    //costruttore  
  
    //metodo  
    Void saluta() {  
        System.out.println("ciao, sono " + nome);  
    }  
}  
  
Public class Main {  
    Public static void main (string[] args) {  
        Persona p = new Persona("Anna", 23);  
        p.saluta(); // chiama il metodo -> "ciao, sono Anna"  
    }  
}
```

Differenza chiave:

- Costruttore = usato solo per inizializzare un nuovo oggetto
- Metodo = usato per far fare qualcosa all'oggetto già esistente

[SOLO C++] Distruttore

Funzione public, ha lo stesso nome della classe ma con prefisso *tilde* ‘~’ (alt + 1 + 2 + 6)

Non ha né parametri né tipi di ritorno

Conviene usarlo quando utilizzo l'heap

Esempio:

```
class Persona {  
    Persona() { cout << "Costruttore chiamato\n"; }  
    ~Persona() { cout << "Distruttore chiamato\n"; }  
};  
  
int Main() {  
    Persona p; // costruttore chiamato  
} // quando esce dallo scope -> distruttore chiamato
```

Static in java e C++

Serve per dire che qualcosa appartiene alla classe e non all'oggetto.

Significa che quel membro o metodo è condiviso da tutte le istanze della classe:

- variabile static / campo static → unica copia comune a tutti gli oggetti

```
class Persona {  
    static int contatore = 0; // campo static  
    Persona() { contatore++; }  
}
```

ATTENZIONE: in Java spesso i termini variabile e campo si usano quasi come sinonimi, ma in realtà il campo è un attributo dichiarato dentro la classe, mentre la variabile è un termine più generico che può indicare una variabile locale (dentro un metodo), un parametro di metodo oppure un campo (attributo di classe o di istanza).

- metodo static → si può chiamare senza creare un oggetto: *classe.nomeMetodo();*

```
class Matematica {  
    static int somma(int a, int b) { return a + b; }  
}  
Int x = Matematica.somma(2, 3); // chiamato senza new
```

In C++ static ha anche altri usi (variabili statiche locali, visibilità limitata nei file)

Modificatore	Dove si usa	Significato
public	campi, metodi, classi	visibile ovunque
private	campi, metodi	visibile solo nella classe
protected	campi, metodi	visibile a package + sottoclassi
(default)	campi, metodi	visibile solo a package
static	campi, metodi	appartiene alla classe, non agli oggetti
final	variabili, metodi, classi	non modificabile / non ereditabile
abstract	classi, metodi	obbligo di implementazione da sottoclassi
synchronized	metodi	un solo thread alla volta
volatile	campi	visibilità immediata tra thread

Aspetto	Java	C++
Ereditarietà multipla	✗ Non supportata (usa interfacce)	✓ Supportata (più classi base)
Gestione memoria	Garbage Collector automatico	Manuale (serve delete o smart pointers)
Puntatori	Non visibili all'utente (solo reference)	Sì, puntatori e reference
Tutto in classi	Quasi tutto è dentro classi (anche metodi statici)	Puoi avere funzioni globali fuori dalle classi
Modifieri default	Default → package-private (visibile nello stesso package)	Default → private
Overloading operatori	✗ Non possibile	✓ Possibile

Aspetto	Java	C++
Costruttori	Simili, ma in Java non esistono i distruttori	C++ ha costruttori, distruttori, copy constructor
Metodi virtuali	Tutti i metodi non-static sono virtuali per default	Devi dichiarare virtual per avere polimorfismo dinamico

Classe Object

- In Java, Object è la superclasse di tutte le classi.
- Se non scrivi extends, la tua classe estende automaticamente Object.
- Contiene metodi base come:

◦ `equals() // public boolean equals(Object obj)`

Concettualmente, equals restituisce true se il *contenuto* di due oggetti è uguale (non il *riferimento*, come per l'operatore ==). Si richiede che implementi una relazione di equivalenza, quindi che goda delle proprietà di *riflessività*, *simmetria*, *transitività* e *consistenza*. Deve essere ridefinito in tutte le classi su cui è necessario effettuare confronti.

Il metodo equals della classe Object, non potendo fare alcuna assunzione sulla struttura interna degli oggetti su cui viene invocato, utilizza semplicemente l'operatore == per confrontarli.

Quindi attenzione: se equals viene usato in una classe senza essere ridefinito, non da' errore in esecuzione, ma il risultato potrebbe essere incorretto.

◦ `hashCode()`

◦ `toString() // public String toString()`

restituisce una stringa che puo' essere considerata come la "rappresentazione testuale" dell'oggetto su cui e' invocato (da usare ad esempio nella stampa).

Deve essere ridefinito in ogni classe che lo usa, per ottenere un risultato significativo (tipicamente, di un oggetto si vogliono stampare i valori delle variabili d'istanza, ed eventualmente una intestazione).

Ecco perché qualunque classe appartiene sempre a Object.

Come personalizzare equals() (@Override)

- Regole
 - Riflessività: `x.equals(x)` dev'essere sempre `true`
 - Simmetria: `x.equals(y)` e `y.equals(x)` dev'essere uguale
 - Transitività: se `x.equals(y)` e `y.equals(z)`, allora `x.equals(z)` deve essere `true`
 - Consistenza: se nulla cambia, il risultato dev'essere sempre lo stesso
 - Con `null`: `x.equals(null)` deve restituire `false`
- Struttura base dell'override

`@Override`

```
public boolean equals(Object o) {  
    // 1. Se è lo stesso oggetto → true  
    if (this == o) return true;  
    // 2. Se è null o classi diverse → false  
    if (o == null || getClass() != o.getClass()) return false;  
    // 3. Cast dell'oggetto  
    MiaClasse other = (MiaClasse) o;  
    // 4. Confronto campi  
    return this.attributo1 == other.attributo1 &&  
        Objects.equals(this.attributo2, other.attributo2);  
}
```

Scanner

Lo scanner è una classe di Java che serve a leggere input testuale (da tastiera, file, stringhe, ...).

Si trova nel package `java.util`, va importata → `import java.util.Scanner;`

Nel caso più comune (leggere da tastiera) si scrive:

```
Scanner sc = new Scanner(System.in);
```

Esempi utili:

```

1. Scanner sc = new Scanner(System.in);
System.out.print("Inserisci una frase: ");
String frase = sc.nextLine();

2. Scanner sc = new Scanner(System.in);
System.out.print("Inserisci una numero intero: ");
int n = sc.nextInt();

3. Scanner sc = new Scanner(System.in);
System.out.print("Inserisci una numero decimale: ");
double d = sc.nextDouble();

4. Scanner sc = new Scanner(System.in);
System.out.print("Inserisci una numeri (0 per finire): ");
while(sc.nextInt()) {
    int n = sc.nextInt();
    if(n == 0) break;
    System.out.println("Hai inserito: " + n);
}

```

Simbolo ":" in Java

In Java lo trovi nel costrutto del for-each:

```

for(Tipo variabile : collezione) {
    // codice che usa variabile
}

```

Si legge “per ogni elemento della collezione”

Esempio:

```

int[] numeri = {1, 2, 3};
for(int n : numeri) {
    System.out.println(n);
}

```

[C++] Posizione di const

ATTENZIONE: “const” si usa in C++, in Java si usa “final” ma in modo diverso!

- **const** int f();

significa che la funzione ritorna un valore costante.

Il valore che ricevi non lo puoi modificare se è un riferimento o un puntatore.

Esempio:

```
const string& getName();
```

- **void f(const int x);**

Significa che il parametro è costante dentro la funzione.

Se provi a scrivere x = 5 → errore

Esempio:

```
void stampa(const string& s){  
    cout << s << endl;      //posso leggerla  
    //s = "ciao";      //errore  
}
```

- **int f() const;**

Questo riguarda i metodi delle classi, significa: “questo metodo non può modificare i campi (variabili membro) dell’oggetto”

Esempio:

```
class Persona{  
    string nome;  
public:  
    string getNome() const{  
        return nome;  
    }  
    void setNome(string nuovo){  
        nome = nuovo;  
    }  
};
```

Se provassi:

```
string getNome() const{  
    nome = "altro";      //ERRORE  
    return nome;  
}
```

Scope (ambito)

Lo **scope** è l’area del codice in cui una variabile è **visibile** e **valida**.

- **Quando una variabile esce dallo scope**, la sua memoria viene automaticamente liberata (se è in **stack**, vedi sotto).
- Questo vale per variabili locali e oggetti creati “senza new”.

Stack vs Heap

La memoria di un programma C++ è divisa in zone, le più importanti per noi sono:

- **Stack** → gestito automaticamente
- **Heap** → gestito manualmente

Stack

- Memoria **automatica**
- Usata per variabili locali, parametri di funzione, return temporanei
- Viene liberata **automaticamente** alla fine dello scope
- Molto veloce, ma dimensione limitata

Heap

- Memoria **dinamica**
- Devi gestirla tu (o con smart pointer)
- Usata con new / new[] per creare oggetti che vivono oltre lo scope corrente
- Più lenta dello stack ma più flessibile (puoi creare oggetti grandi e decidere quanto restano in vita)
- Devi liberarla con delete / delete[] o smart pointers

Relazione con i distruttori

- Oggetti nello **stack** → il distruttore viene chiamato automaticamente quando escono dallo scope.
- Oggetti nell'**heap** → il distruttore viene chiamato solo quando usi delete (o lo smart pointer decide di distruggere l'oggetto).

Vita variabili

- **Allocazione statica:** deallocazione al termine del programma -> DATA SEGMENT
- **Allocazione automatica:** deallocazione al termine del blocco -> STACK
- **Allocazione dinamica:** deallocazione tramite l'operatore delete -> HEAP

Vita oggetti: visti come un'allocazione automatica (è presente un distruttore), tempo di esecuzione del blocco

Overloading operatore di assegnamento

Va fatto altrimenti due oggetti con variabili allocate dinamicamente punteranno allo stesso indirizzo in memoria.

L'overload dell'operatore di assegnazione in C++ (e in altri linguaggi) permette di definire il comportamento dell'operatore "=" quando viene utilizzato con tipi di dati definiti dall'utente. In pratica, si può ridefinire il modo in cui gli oggetti di una classe vengono assegnati ad altri oggetti della stessa classe o di tipi diversi.

Quando si lavora con puntatori o risorse dinamiche all'interno di una classe, l'assegnazione di default potrebbe non essere sufficiente o potrebbe portare a problemi di memoria (ad esempio, copie superficiali di puntatori, perdite di memoria). L'overload permette di implementare una copia profonda, allocando nuova memoria per i dati e copiando i contenuti.

Costruttore di copia

È un tipo speciale di costruttore che crea un nuovo oggetto copiando un oggetto già esistente.

Se **non lo scrivi tu**, il compilatore crea un **costruttore di copia predefinito** che:

- copia **membro per membro** (shallow copy)
- copia i valori, non le risorse collegate (come puntatori)

Vale la stessa cosa per l'overloading dell'operatore di assegnamento; devo copiare attributo per attributo in caso di allocazione dinamica della memoria.

Shallow copy (copia superficiale)

- Copia solo il contenuto delle variabili membro
- Se c'è un puntatore, copia **solo l'indirizzo**, quindi i due oggetti puntano alla **stessa memoria**
- Problema: quando un oggetto viene distrutto e libera la memoria, l'altro resta con un puntatore "morto" → *dangling pointer* o doppio delete

Deep copy (copia profonda)

- Crei una nuova copia della risorsa allocata
- Ogni oggetto ha la **sua** memoria indipendente

La **regola del tre** è una regola empirica di programmazione in C++ per l'implementazione dei metodi predefiniti di una classe destinata alla gestione di risorse. Afferma che, se la classe definisce esplicitamente almeno uno tra distruttore, costruttore di copia e operatore di assegnamento, allora tutti e tre i metodi devono essere definiti esplicitamente.

[C++] Class vector

Implementazione astratta della classe vettore.

Utilizziamo la funzionalità dei template per creare la classe.

Allocchiamo l'array di tipo T (template) dinamicamente -> rispettiamo la regola del tre.

Rappresenta un array dinamico:

- Puoi aggiungere o rimuovere elementi senza preoccuparti delle dimensioni.
- Gestisce in automatico l'allocazione di memoria.
- **Vector():** crea vettore vuoto (size=0, capacity=0)
- **Vector(int n):** crea vettore di dimensione n (size=0, capacity=n)
- **Vector(int n, const T& k):** crea un vettore con n elementi inizializzati col valore k
- **Vector(const vector<T>& V):** costruttore di copia (deep copy)
- **Reserve(int n):** alloca un'area di memoria di dimensione sufficiente a contenere n elementi (e quindi modifica capacity)
- **size():** numero di elementi
- **empty():** true se è vuoto, false altrimenti
- **push_back(x):** aggiunge x in fondo al vettore
- **pop_back():** rimuove l'ultimo elemento del vettore
- **clear():** elimina tutti gli elementi del vettore
- **capacity():** ritorna la quantità di memoria allocata per questo vettore

Standard Template Library (STL)

È un set di classi template che fornisce strutture e funzioni comuni in ambito di programmazione.

Ha quattro componenti: algoritmi, containers, funzioni e iteratori.

Tra i containers più importanti abbiamo:

- Sequentziali:
 - **Vector<T>**
 - **List<T>**
 - **Stack<T>**
- Associativi:

- `Set<T>`
- `MultiSet<T>`
- `Map<T1, T2>`

`List<T>`

Interfaccia che rappresenta una sequenza ordinata di elementi (può contenere duplicati).

Gli elementi hanno un indice (posizione).

Implementazioni più usate:

- `ArrayList<T>` → basata su array, accesso veloce per indice
- `LinkedList<T>` → lista doppiamente concatenata, veloce per inserimenti/rimozioni

Esempio:

```
public class TestList {
    public static void main(String[] args) {
        List<String> lista = new ArrayList<>();

        lista.add("Anna");
        lista.add("Mario");
        lista.add("Anna"); // duplicato ok

        System.out.println(lista.get(1)); // "Mario"
    }
}
```

`Stack<T>`

È una pila (struttura LIFO: Last In, First Out).

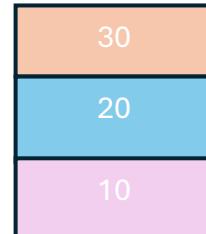
Metodi principali:

- `push(elem)` → aggiunge in cima
- `pop()` → rimuove e restituisce l'elemento in cima
- `peek()` → restituisce l'elemento in cima senza rimuoverlo
- `empty()` → controlla se è vuoto

Esempio:

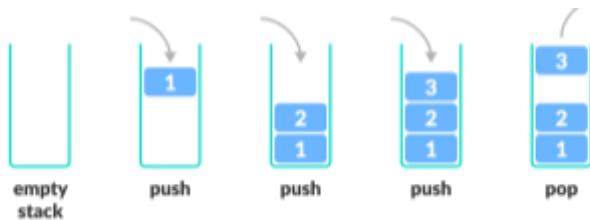
```
public class TestStack {  
    public static void main(String[] args) {  
        Stack<Integer> pila = new Stack<>();
```

```
pila.push(10);  
pila.push(20);  
pila.push(30);
```



```
    System.out.println(pila.pop()); // 30  
    System.out.println(pila.peek()); // 20  
}
```

```
}
```



Set<T>

Un set è una collezione senza duplicati.

T è il tipo degli elementi (es. Set<String>, Set<Integer>)

Le implementazioni più comuni:

- **HashSet** → non garantisce ordine.
- **TreeSet** → mantiene ordine crescente.
- **LinkedHashSet** → mantiene ordine di inserimento.

Operazioni principali:

- **add(elem)** → aggiunge un elemento
- **remove(elem)** → rimuove un elemento
- **contains(elem)** → controlla se l'elemento è presente
- **size()** → numero elementi

Esempio:

```
public class TestSet {
```

```

public static void main(String[] args) {
    // implementazione

    Set<String> nomi = new HashSet<>();

    nomi.add("Anna");
    nomi.add("Mario");
    nomi.add("Anna"); // duplicato, NON aggiunto

    System.out.println("Contiene Luca? " +
        nomi.contains("Luca")); // false

    System.out.println("Dimensione: " + nomi.size()); // 2
}

}

```

Modo 1 – forma abbreviata (solo in dichiarazione)

```

int[] arr = {1, 2, 3};
int arr[] = {1, 2, 3}; // parentesi dopo il nome → stessa cosa

```

- Funziona **solo** quando dichiari e inizializzi insieme.
- È una scorciatoia che il compilatore trasforma internamente in new int[]{...}.

Modo 2 – forma esplicita con new

```
int[] arr = new int[]{1, 2, 3};
```

```
int arr[] = new int[]{1, 2, 3}; // stessa cosa, parentesi dopo il nome
```

- Funziona sia in dichiarazione che in assegnazione successiva:

```

int[] arr;
arr = new int[]{1, 2, 3};

```

- Serve obbligatoriamente se l'inizializzazione avviene dopo.

Primitive

- Sono i **tipi base** già integrati nel linguaggio.

- Contengono **direttamente** il valore.
- In **Java**: int, double, float, char, boolean, byte, short, long.
- In **C++**: int, double, float, char, bool, ecc.
- Quando li copi o li passi a una funzione, si **duplica il valore** (non sono collegati tra loro).

Esempio:

```
int a = 5;
int b = a; // b = 5, indipendente da a
a = 10; // b resta 5
```

Reference

- Non contengono direttamente il valore, ma **l'indirizzo** di un oggetto in memoria.
- In **Java**: tutte le classi, array, String, List, ecc.
- In **C++**: puntatori (*) e riferimenti (&).
- Quando li copi o li passi a una funzione, **puntano allo stesso oggetto** → se lo modifichi, cambia per tutti.

Esempio:

```
int[] x = {1, 2, 3};
int[] y = x; // y punta allo stesso array di x
y[0] = 99; // anche x[0] diventa 99
```

Quindi:

- Primitive -> valore diretto, indipendente
- Reference -> collegamento a un oggetto, condiviso

Classi wrapper

I costruttori delle classi wrapper permettono di creare oggetti dei tipi primitivi.

Dato un tipo primitivo si ottiene il corrispondente Data Object usando la prima lettera maiuscola:

- **byte** -> Byte
- **short** -> Short

- int -> Integer
- long -> Long
- float -> Float
- double -> Double
- char -> Character
- boolean -> Boolean

Autoboxing: sintassi semplificata per nascondere la creazione degli oggetti wrapped:

```
Integer x = 3 -> Integer x = new Integer(3)
```

Sono sempre delle reference, allocati dinamicamente nello heap.

Sono immutabili come la classe String: quando faccio un'operazione di assegnamento l'oggetto non viene sovrascritto ma ne viene creato uno nuovo -> garbage collector

Unboxing: accesso ai valori degli oggetti `int y = x.intValue()`

Su Java unboxing e autoboxing vengono gestiti automaticamente.

Ereditarietà in Java

- **Parola chiave: extends**

```
class Animale { }

class Cane extends Animale { }
```

- Una classe può ereditare **da una sola classe** (solo ereditarietà singola, no multipla).
- Tutte le classi ereditano implicitamente da Object.
- Supporta il **polimorfismo dinamico**: i metodi possono essere sovrascritti (@Override).
- **super**: parola chiave che serve per riferirsi alla superclasse dell'oggetto corrente, viene usata in due modi principali:
 - **richiamare il costruttore della superclasse**
Se una classe eredita dall'altra, puoi usare `super(...)` nel costruttore per chiamare il costruttore della superclasse e inizializzare i campi ereditati
 - **accedere a metodi o campi della superclasse**
Se la classe figlia ha un metodo o un campo con lo stesso nome della superclasse, puoi usare `super.metodo()` o `super.campo` per richiamare quello della superclasse, evitando l'override

Esempio:

```
class Persona {
```

```

        string nome;
    Persona(string nome) { this.nome = nome; }
    Void saluta() {
        System.out.println("ciao, sono " + nome);
    }
}

Class Studente extends Persona {
    Int matricola;
    Studente(string nome, int matricola) {
        Super(nome); // chiama il costr. della superclasse Persona
        this.matricola = matricola;
    }
    @Override
    Void saluta() {
        Super.saluta(); // chiama il metodo saluta() della supercl.
        System.out.println("la mia matricola è " +
        matricola);
    }
}

Public class Main {
    Public static void main(String[] args) {
        Studente s = new Studente("Mario", 12345);
        s.saluta();
    }
}

```

- **Modificatori di visibilità:**

- public → visibile ovunque
- protected → visibile a sottoclassi e stesso package

- (default, senza nulla) → visibile solo nello stesso package
- private → visibile solo dentro la classe

In Java non esiste ereditarietà multipla tra classi, ma si può simulare con **interfacce** (implements).

	Classe in cui è dichiarata	Sottoclassi	All'esterno della classe	Nello stesso package
public	✓	✓	✓	✓
private	✓	✗	✗	✗
protected	✓	✓	✗	✓
package	✓	✗	✗	✓

Ereditarietà in C++

- **Parola chiave:** “`:`” con tipo di ereditarietà

```
class Animale { };

class Cane : public Animale { };
```

- Una classe può ereditare da **più classi contemporaneamente** (ereditarietà multipla).
- Non c'è una classe universale tipo Object.
- Supporta **polimorfismo dinamico** solo con funzioni virtual.
- **super non esiste**, ma si usano i nomi esplicativi con `::` per accedere a metodi/costruttori della base.
- Tipi di ereditarietà:
 - public → i membri pubblici/protetti restano tali
 - protected → i membri pubblici diventano protetti
 - private → i membri pubblici/protetti diventano privati

Tema	Java	C++
Ereditarietà multipla	✗ No (solo 1 classe base, ma più interfacce)	✓ Sì

Tema	Java	C++
Classe universale	Tutto eredita da Object	Non c'è una radice comune
Parola chiave	extends	: public/protected/private
Polimorfismo	Sempre dinamico (metodi override)	Solo se i metodi sono virtual
Costruttori	Con super()	Con : Base(...)
Modificatori	public, protected, private, package	public, protected, private (senza package)

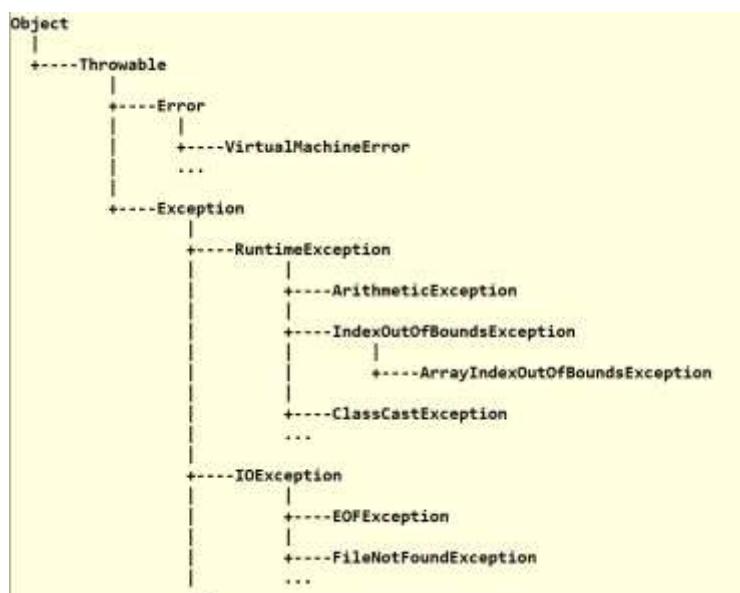
Eccezioni in Java

In Java, un'**eccezione** è un evento che interrompe il flusso normale del programma quando succede qualcosa di imprevisto.

Java non “ignora” gli errori: se accade qualcosa di imprevisto (divisione per zero, accesso a un file inesistente, variabile null, ecc.), viene generata un'**eccezione**.

- Se l'eccezione non viene gestita → il programma si interrompe con un **errore a runtime**.
- Se l'eccezione viene gestita → il programma continua a funzionare in modo controllato.

Gerarchia base delle eccezioni:



- **throw**: Serve a lanciare un'eccezione in un punto preciso, dopo throw devi sempre passare un oggetto che sia un'eccezione (**Exception** o **RuntimeException**)

```

public class ThrowExample {

    public static void checkAge(int age) {
        if(age < 18) {
            throw new aMinorException("a minoooorr");
        }
        System.out.println("Accesso consentito. ");
    }
}

```

- **try:** Qui metti il codice che potrebbe generare un'eccezione. È il “blocco a rischio”, serve a dire al compilatore e al runtime “attenzione, qui dentro potrebbe accadere qualcosa di pericoloso”.

```

public class TryExample {

    public static void main(int age) {
        try {
            int risultato = 10 / 0;
            System.out.println("Risultato: " +
                risultato);
        } catch(ArithemticException e) {
            System.out.println("Divisione per zero!");
        }
    }
}

```

- **catch:** Segue il try e serve a intercettare e gestire l'eccezione. Puoi avere più catch per tipi diversi di eccezioni. Dentro il catch puoi decidere come reagire: stampare un messaggio, correggere il problema, ecc.

```

public class CatchExample {

    public static void main(String[] args) {
        try {
            String s = null;
            System.out.println(s.length());
        } catch(NullPointerException e) {

```

```

        System.out.println("oggetto null");
    } catch(Exception e) {
        System.out.println("errore generico");
    }
}
}

```

- **finally:** È opzionale e viene eseguito sempre, **sia che ci sia un'eccezione sia che non ci sia**. Serve per operazioni di pulizia, come chiudere file o connessioni, assicurandoti che avvengano comunque. Esempio chiudere scanner: `scanner.close()`

```

public class FinallyExample {

    public static void main(String[] args) {
        try {
            int[] numeri = {1, 2, 3};
            System.out.println(numeri[5]);
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Errore: indice");
        } finally {
            System.out.println("Questo messaggio viene
SEMPRE stampato!");
        }
    }
}

```

Interfacce in Java

Un'interfaccia in Java è come un contratto: definisce cosa una classe deve fare, ma non come.

- Dentro un'interfaccia ci sono solo metodi astratti (senza corpo) e costanti (variabili `public static final`).
- Una classe che “implementa” un'interfaccia è obbligata a fornire il codice (l'implementazione) per tutti i metodi dichiarati.

- Una classe può implementare più interfacce (mentre può estendere una sola classe). Questo permette di aggirare il limite dell'ereditarietà singola.
- Dalla versione Java 8, nelle interfacce possono esserci anche metodi default (con un corpo) e metodi statici.

In breve:

- classi astratte = possono avere metodi già implementati e stati (variabili).
- interfacce = servono per definire regole comuni, senza preoccuparsi di come verranno realizzate.

Esempio:

```
public interface Predatore {
    void caccia();
}

Public interface Preda {
    Void scappa();
}

Public class Gazzella implements Preda {
    @Override
    Public void scappa() {
        System.out.println("la gazzella si nasconde");
    }
}

Public class Leone implements Predatore {
    @Override
    Public void caccia() {
        System.out.println("il leone parte all'attacco");
    }
}

Public class Pesce implements Preda, Predatore {
    @Override
    Public void caccia() {
```

```

        System.out.println("il pesce grande caccia quelli più
piccoli");
    }

    @Override

    Public void scappa() {

        System.out.println("il pesce piccolo scappa da quelli
più grandi");
    }

}

Public class Main {

    Public static void main(string[] args){

        Leone leone = new Leone();

        Gazzella gazzella = new Gazzella();

        Pesce pesce = new Pesce();

        leone.caccia();           // il leone parte all'attacco

        gazzella.scappa();       // la gazzella si nasconde

        pesce.scappa();          // il pesce piccolo scappa da quelli più grandi

        pesce.caccia();          // il pesce grande caccia quelli più piccoli
    }

}

```

Tipi generici

Immagina di voler fare una scatola (Box) che può contenere qualcosa.

Se la programmi “senza generics”, devi decidere da subito cosa ci metti dentro (solo String, solo Integer, ecc.), oppure usare Object, che però è poco sicuro.

Con i generics dici: “Questa scatola può contenere solo oggetti di tipo X (che decido io quando creo la scatola)”.

I generics servono per scrivere classi e metodi riutilizzabili che funzionano con tipi diversi, mantenendo la sicurezza dei tipi.

Senza generics:

```
ArrayList list = new ArrayList();
```

```
list.add("Ciao");  
String s = (String) list.get(0); // serve il cast
```

Con generics:

```
ArrayList<String> list = new ArrayList<>();  
list.add("Ciao");  
String s = list.get(0); // niente cast, più sicuro
```

Puoi anche definire classi generiche:

```
class Box<T> { // T è un "tipo generico"  
    private T value;  
    public void set(T value) { this.value = value; }  
    public T get() { return value; }  
}
```

```
Box<Integer> b = new Box<>();  
b.set(42);  
System.out.println(b.get()); // stampa 42
```

Un modo per pensarli: i generics sono come un parametro per il tipo, esattamente come i parametri di un metodo, ma applicati ai tipi invece che ai valori.

Il <T> indica un tipo generico.

- Quando vedi `HashSet<T>`, significa che quella classe può lavorare con oggetti di qualsiasi tipo, e sarai tu a specificarlo.
- T è solo un nome convenzionale (potrebbe essere <E>, <K, V> se sono più tipi).

Iterable e iterator

Iterable

Un oggetto che implementa l’interfaccia Iterable può essere “iterato”, cioè puoi scorrerlo in un ciclo for-each.

Tutte le Collection (come List, Set, ecc.) implementano Iterable.

```
Iterator<T> iterator();
```

Iterator

L'Iterator è l'oggetto che effettivamente ti permette di scorrere gli elementi uno alla volta.

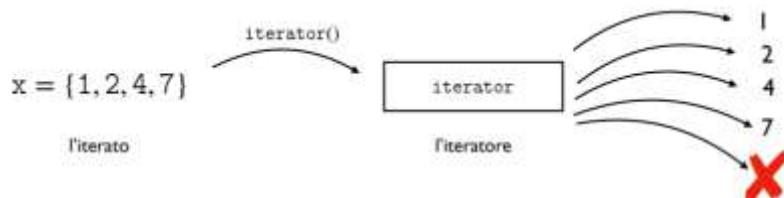
Ha tre metodi principali:

- `hasNext()` → dice se ci sono ancora elementi da leggere.
- `next()` → restituisce l'elemento successivo.
- `remove()` → (opzionale) rimuove l'ultimo elemento letto.

Iterable = interfaccia che dice “puoi scorrere questo oggetto con un for-each”.

Iterator = oggetto che sa scorrere passo passo gli elementi (con `hasNext()` e `next()`).

Iterable e Iterator



```
public interface java.lang.Iterable<T> {
    Iterator<T> iterator();
}

public interface java.util.Iterator<E> {
    boolean hasNext();
    E next();
}
```

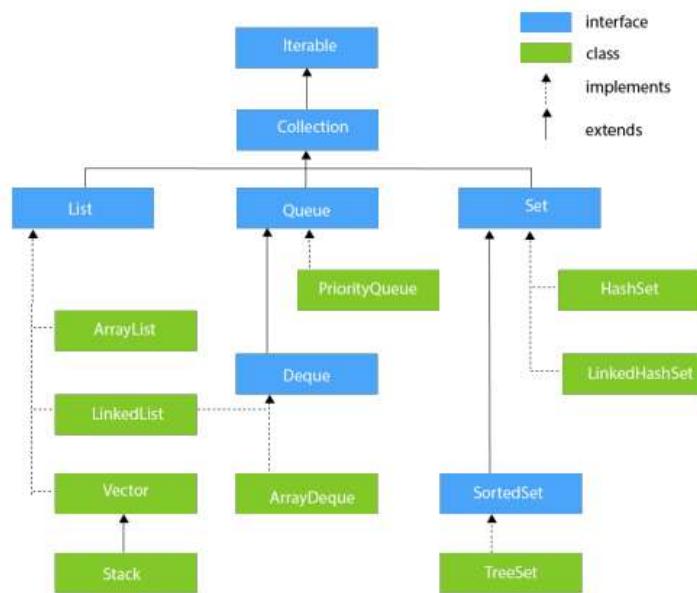
Collection framework

un framework è un insieme di strumenti, librerie e linee guida che forniscono una struttura e delle convenzioni per lo sviluppo di software, su cui i programmati possono costruire applicazioni, mentre un'interfaccia è il componente (fisico o logico) che permette a diversi sistemi di comunicare ed interagire.

- Equivalenti alla STL di C++
- Insieme di classi e interfacce già pronte in Java per gestire insiemi di oggetti (liste, insiemi, mappe..)
 - Distinte in base a come vengono raggruppati gli elementi
 - Distinte in base alla modalità d'accesso
- All'interno del package `java.util`

Collection framework

Gerarchia



Interfaccia collection

- È l'interfaccia base di tutte le collezioni.
- Definisce operazioni comuni come:
 - `add(E e)` → aggiunge un elemento
 - `remove(Object o)` → rimuove
 - `size()` → numero di elementi
 - `isEmpty()` → controlla se vuota
 - `iterator()` → per scorrerla con un Iterator

Tutte le collezioni (eccetto `Map`) ereditano da `Collection`

Collection framework

L'interfaccia Collection

```
public interface Collection<E> extends Iterable<E> {
    int size();
    boolean isEmpty();
    boolean contains(Object o);
    Iterator<E> iterator(); // from Iterable<E>
    boolean add(E e);
    boolean remove(Object o);
    void clear();
    boolean equals(Object o);
    int hashCode();
}
```

Interfaccia set

- È una sottoclasse di collection.
- Rappresenta un insieme senza duplicati.
- Non garantisce ordine (a meno che tu usi implementazioni particolari, tipo TreeSet).

Collection framework

L'interfaccia Set

```
public interface Set<E> extends Collection<E> {
    int size();
    boolean isEmpty();
    boolean contains(Object o);
    Iterator<E> iterator();
    boolean add(E e);
    boolean remove(Object o);
    void clear();
    boolean equals(Object o);
    int hashCode();
}
```

Classe HashSet<T>

È una delle implementazioni più comuni di Set.

- Usa una tabella hash per memorizzare gli elementi.
- Vantaggi: inserimento, ricerca e rimozione sono molto veloci (complessità O(1) in media).
- Non mantiene ordine sugli elementi.

Esempio:

```
HashSet<String> set = new HashSet<>();
set.add("ciao");
set.add("mondo");
```

```
set.add("ciao");           //ignorato, niente duplicati  
System.out.println(set);    // [ciao, mondo] (ordine casuale)
```

Metodo hashCode()

Quando usi strutture come *HashSet* o *HashMap*, Java deve capire dove mettere l'oggetto nella tabella hash.

Per farlo, usa il metodo *hashCode()* di ogni oggetto.

Regola:

- Oggetti uguali secondo *equals()* devono avere lo stesso *hashCode()*.
- Non è obbligatorio il contrario (hash uguali non garantiscono oggetti uguali → si chiama “collisione”).

```
String s1 = "ciao";  
String s2 = "ciao";  
System.out.println(s1.hashCode()); //stesso numero  
System.out.println(s2.hashCode()); //stesso numero
```

Classe Vector<T>

- È simile ad un *ArrayList*, cioè una lista che cresce dinamicamente.
- La differenza: *Vector* è sincronizzato → sicuro nei thread multipli, ma più lento.
- Oggi si preferisce *ArrayList*, a meno che non serva la sincronizzazione.

Esempio:

```
Vector<Integer> v = new Vector<>();  
v.add(1);  
v.add(2);  
System.out.println(v);      // [1, 2]
```

Classe HashMap<K, V>

- È una struttura che memorizza coppie chiave → valore.
- Non permette chiavi duplicate.
- Usa anche lei la tabella hash.

- L'accesso a un valore tramite la chiave è molto veloce.

<K, V>:

- K = tipo della chiave
- V = tipo del valore

Esempio:

```
HashMap<Integer, String> map = new HashMap<>();
map.put(1, "Ciao");
map.put(2, "Mondo");

System.out.println(map.get(1));           // Ciao
```

RICAPITOLANDO:

- **<T>** = **tipo generico** (deciso quando crei l'oggetto). *Sempre usati, fondamentali*
- **Collection** = interfaccia base per insiemi di oggetti. *Usatissime*
- **Set** = sotto-interfaccia che vieta i duplicati.
- **HashSet<T>** = implementazione di Set basata su tabella hash. *Usato quando serve unicità*
- **hashCode()** = usato per posizionare gli oggetti nelle strutture hash. *Usato quando serve unicità*
- **Vector<T>** = lista sincronizzata, simile ad ArrayList. *Oggi meno usata*
- **HashMap<K, V>** = mappa chiave → valore, basata su hash.

Classi astratte

Una classe astratta è una classe che:

- non può essere istanziata direttamente (non puoi fare *new ClasseAstratta()*),
- serve come modello/base per altre classi,
- può contenere sia metodi astratti (senza corpo) sia metodi concreti (con implementazione).

In sintesi: ti permette di definire una struttura comune che le sottoclassi devono completare.

Si usa:

- Quando hai concetti generali che hanno parti comuni e parti specifiche da definire nelle sottoclassi.

- Esempio: un Animale ha sempre un nome, può dormire, ma il verso cambia a seconda che sia un cane o un gatto.

Caratteristiche importanti

- Può avere costruttori (anche se non viene mai istanziata direttamente).
- Può avere variabili di istanza e metodi concreti.
- Le sottoclassi devono implementare i metodi astratti, altrimenti diventano astratte pure loro.
- Può estendere un'altra classe e implementare interfacce.

Differenza con interfacce

- Interfaccia: è solo un contratto, non può avere stato (tranne costanti), e di solito contiene metodi astratti (più default e statici da Java 8).
- Classe astratta: può avere stato (variabili di istanza), costruttori, e metodi sia concreti che astratti.

Si usa **classe astratta** quando vuoi dare anche un po' di implementazione comune, non solo un "contratto".

Esempio:

```
// Classe astratta
abstract class Animale {
    String nome;
    Animale(String nome) {
        this.nome = nome;
    }
    // Metodo astratto (le sottoclassi DEVONO implementarlo)
    abstract void verso();
    // Metodo concreto (già implementato)
    void dormi() {
        System.out.println(nome + " sta dormendo... Zzz");
    }
}
```

```
// Sottoclasse concreta
class Cane extends Animale {
    Cane(String nome) {
        super(nome);
    }
    void verso() {
        System.out.println("Bau!");
    }
}

// Sottoclasse concreta
class Gatto extends Animale {
    Gatto(String nome) {
        super(nome);
    }
    void verso() {
        System.out.println("Miao!");
    }
}

public class Main {
    public static void main(String[] args) {
        Animale a1 = new Cane("Fido");
        Animale a2 = new Gatto("Micia");

        a1.verso(); // Bau!
        a2.verso(); // Miao!
        a1.dormi(); // Fido sta dormendo... Zzz
    }
}
```

}

Differenza tra parametro della funzione e classi di appartenenza

- Parametro della funzione: conta chi può essere accettato come input.

“[C++] Si considerino le classi **Forma**, **Rettangolo** e **Quadrato**. La classe **Rettangolo** è derivata da **Forma**, mentre la classe **Quadrato** è derivata da **Rettangolo**. La seguente funzione

```
double shift(Rettangolo r) {...}
```

può accettare come argomenti oggetti...”

Se la funzione accetta un rettangolo, non può ricevere una super classe (forma), perché non è detto che una forma abbia i dettagli del rettangolo

Può invece ricevere una sottoclasse (quadrato), perché un quadrato è anche un rettangolo

- Classi di appartenenza: conta a quali classe appartiene un oggetto lungo la catena dell'ereditarietà.

“[Java] Si considerino le classi **Forma**, **Rettangolo** e **Quadrato**. La classe **Rettangolo** è derivata da **Forma**, mentre la classe **Quadrato** è derivata da **Rettangolo**. Le classi di appartenenza della classe Rettangolo sono...”

Rettangolo appartiene a sé stesso e a tutte le sue superclassi (forma e Object)

Non appartiene alle sue sottoclassi (quadrato), perché non vale al contrario: un rettangolo generico non è sempre un quadrato.

Attributo d'istanza e attributo di classe

in breve: istanza = legato al singolo oggetto

classe = unico per tutti

Attributo di istanza

- Dichiарато senza static.
- Ogni oggetto della classe ha la sua copia personale.

Esempio:

```
class Rettangolo {
```

```

    int base; // attributo d'istanza

    int altezza; // ogni Rettangolo ha la sua base e la sua altezza
}

```

Attributo di classe

- Dichiara con static.
- Esiste una sola copia condivisa da tutta la classe, indipendentemente da quante istanze crei

Esempio:

```

class Rettangolo {

    static int contatore = 0; // attributo di classe

    Rettangolo() { contatore++; }

}

```

Qui **contatore** conta quante istanze sono state create: è unico per tutta la classe

Classi e i loro comandi utili

Classe	Metodo/comando	Descrizione
ArrayList / List	add(element)	Aggiunge un elemento alla lista
	get(index)	Restituisce l'elemento in posizione index
	set(index, element)	Sostituisce l'elemento in posizione index
	remove(index) / remove(object)	Rimuove un elemento (per posizione o valore)
	size()	Restituisce il numero di elementi
	contains(object)	Verifica se un elemento è presente
	isEmpty()	Controlla se la lista è vuota
	clear()	Rimuove tutti gli elementi
HashSet / Set	add(element)	Aggiunge un elemento (evita duplicati)
	remove(object)	Rimuove l'elemento se presente

	contains(object)	Verifica se l'elemento è presente
	size()	Numero di elementi nel set
	isEmpty()	Controlla se è vuoto
	clear()	Rimuove tutti gli elementi
Iterator	hasNext()	Ritorna true se ci sono ancora elementi da scorrere
	next()	Restituisce il prossimo elemento
	remove()	Rimuove l'ultimo elemento restituito da next()
Object	equals(obj)	Confronta due oggetti (default: confronto riferimenti, ma si può overrideare)
	hashCode()	Restituisce un codice intero usato nelle strutture hash (da overrideare con equals)
	getClass()	Restituisce la classe effettiva dell'oggetto in esecuzione

DOMANDE DIFFICILI DEL PARZIALE

• {

[C++] Si considerino le classi X, Y e Z. La classe Y è derivata da X, mentre la classe Z è derivata da Y. La seguente funzione

X foo(Y obj) const {...}

Può accettare come argomenti oggetti

- di tipo Z e Y
- di tipo Y e Z
- di tipo X, Y e Object
- di tipo Y
- il codice non compila

}

Ragionamento passo passo

Parametro formale: `Y obj` → significa che la funzione prende in input un oggetto di tipo `Y`, passato per valore.

Quindi:

- Posso passare un oggetto `Y`
- Posso passare un oggetto `Z`? → sì perché `Z` è un `Y` (`Z` estende `Y` → conversione *upcasting* possibile)
- Posso passare un oggetto `X`? No → `X` è superclasse, e un oggetto della superclasse non può essere automaticamente trasformato in sottoclasse (servirebbe un cast esplicito e comunque non sicuro).

Risposta giusta: la funzione accetta oggetti di tipo `Y` e `Z`

Suggerimento: in C++, quando vedi un parametro come `Y obj`, ricorda:

1. Puoi passare `Y` o sottoclassi (`Z`)
2. Non puoi passare la superclasse (`X`)

- {

[Java] Si considerino le classi `A`, `B` e `C`. Le classi `C` e `B` estendono `A`. La classe `A` definisce un metodo `foo` che viene sovrascritto sia dalla classe `B` che dalla classe `C`. Si consideri il seguente frammento di codice.

```
A obj = new C()  
((B) obj).foo();
```

- viene invocato il metodo `foo` definito nella classe `A`
- viene invocato il metodo `foo` definito nella classe `B`
- viene sollevata una `ClassCastException`
- viene rilevato un errore a tempo di compilazione
- nessuna delle precedenti

```
}
```

Ragionamento passo passo

- Dichiarazione e istanziazione:

```
A obj = new C();
```

- `obj` ha tipo statico `A` (quello dichiarato a sinistra).

- **obj** ha tipo dinamico C (quello creato con new).

- Cast:

(B) obj

- Tu stai dicendo: "tratta obj come se fosse un B".
- Ma in realtà obj è un C → e C non è sottotipo di B, perché B e C sono fratelli, entrambi derivano da A.
- Quindi questo cast è illegale a runtime → genera una **ClassCastException**.

- Se non ci fosse il cast (solo **obj.foo()**):
- Polimorfismo dinamico: avrebbe chiamato il metodo **foo** di C, perché l'oggetto è di tipo C a runtime.

Risposta giusta: con il cast **(B) obj**, a runtime viene lanciata una **ClassCastException**

Suggerimento: in Java, quando vedi cast e override, ricorda:

1. Il metodo scelto dipende sempre dal tipo dinamico (polimorfismo)
2. Se il cast è incompatibile (qui C non è un B), scatta **ClassCastException**

- {

[Java] Si considerino le classi A, B e C. La classe C è derivata da B, mentre la classe B è derivata da A. La classe A definisce un metodo **foo** che la classe B ridefinisce mentre la classe C non ridefinisce. Si consideri il seguente frammento di codice.

```
A obj = new C()
    Obj.foo();
```

viene invocato il metodo **foo** definito nella classe A

viene invocato il metodo **foo** definito nella classe B

viene sollevata una **ClassCastException**

viene rilevato un errore a tempo di compilazione

nessuna delle precedenti

}

Ragionamento passo passo

- Dichiarazione e istanza:

```
A obj = new C();
```

- Tipo statico: A (quello dichiarato a sinistra)
- Tipo dinamico: C (oggetto reale creato).

- Chiamata a **foo()**:

- In Java, il metodo invocato si decide a runtime in base al tipo dinamico (polimorfismo)
- Obj è un C
- C non ridefinisce **foo**, quindi guarda nella sua superclasse (B)
- B ridefinisce **foo**, quindi viene chiamato il **foo** di B

Risposta giusta: viene invocato il metodo **foo** definito in B

- {

[C++] Si consideri il seguente programma

```
Void f() {
    Throw 1;
}

Int main() {
    Try {
        f();
    } catch(int x) {
        Cout << "1" << endl;
    } catch(string y) {
        Cout << "2" << endl;
    } catch(Razionale z) {
        Cout << "3" << endl;
    }
    Return 0;
}
```

- 1
- 1 2
- 1 2 3
- nessuna delle precedenti
- }

Cosa succede

1. Main chiama **f()**
2. Dentro **f()** viene lanciata un'eccezione con **throw 1;** → qui viene lanciato un int
3. Torniamo al **try**
 - Il primo **catch** controlla se l'eccezione è di tipo int: sì! Quindi entra lì
 - Stampa "1"
 - Dopo aver trovato un catch valido, gli altri vengono ignorati

Perché non gli altri?

- **Catch(string y)** non viene eseguito perché lancia un int, non una string
- **Catch(Razionale z)** idem, il tipo non corrisponde

