

CORREZIONE TEORIA PA

```
1 static void foo(String... s) {
2     System.out.println(s.length);
3 }
4
5 public static void main(String[] args) {
6     foo("Ciao");
7 }
```

CHATGPT

Il codice fornito sembra essere scritto in Java, non in C++. La funzione foo è dichiarata con un parametro varargs String... s, il che significa che può accettare un numero variabile di argomenti di tipo String.

Nel metodo main, viene chiamata la funzione foo con un singolo argomento di tipo String, ovvero "Ciao". Questo è perfettamente valido in Java, e il compilatore Java convertirà automaticamente l'argomento in un array di stringhe.

Il risultato di questo codice sarà la stampa della lunghezza dell'array di stringhe, che è 1, perché è stato passato un solo argomento. Quindi, l'output sarà:

1

In sintesi, il codice definisce una funzione foo che accetta un numero variabile di stringhe come argomento e stampa la lunghezza dell'array di stringhe passato come argomento. Nel metodo main, viene chiamata foo con un singolo argomento, il cui risultato è la stampa di "1" a causa della lunghezza dell'array di stringhe contenente un elemento.

- stampa 1
- dà errore in compilazione: avrei dovuto scrivere "String s" anziché "String... s"
- stampa 4
- dà errore in compilazione: avrei dovuto scrivere System.out.println(s.length());

```
1 ArrayList<Integer> li = new ArrayList<Integer>();
2 li.add(1); li.add(2); li.add(3);
3 stampaLista(li);
4
5
```

Il motivo per cui il codice stampa "123" è legato al tipo del parametro list nella funzione stampaLista e al fatto che gli elementi della lista li sono di tipo Integer.

Nel tuo codice, li è una lista di oggetti Integer. Quando passi li come argomento alla funzione stampaLista, stai utilizzando una wildcard <? extends Comparable<?>> per dichiarare il tipo del parametro. Questa wildcard indica che la funzione può accettare una lista di oggetti di qualsiasi tipo che sia una sottoclassificazione di Comparable<?>. Gli oggetti Integer soddisfano questa condizione poiché Integer implementa l'interfaccia Comparable<Integer>.

All'interno della funzione stampaLista, vengono iterati gli elementi della lista e stampati. Poiché stampaLista non specifica un tipo particolare per gli elementi della lista, vengono trattati come oggetti generici (Object). Poiché li contiene oggetti Integer, vengono stampati i valori 1, 2 e 3.

Quindi, il risultato dell'esecuzione della funzione stampaLista(li) sarà la stampa di "1 2 3" perché vengono iterati e stampati gli oggetti Integer all'interno della lista li.

List<Integer> stampa 123

ArrayList<Object> stampa errore

List<? super Number> stampa errore

List<? implements Serializable> stampa errore

List<?> stampa 123

List<? extends Number> stampa 123

List<Number> stampa errore

N.B. la 7 è corretta

Ecco una valutazione di ciascuna delle dichiarazioni in base alla correttezza:

1. ArrayList<Integer> v = new List<Integer>();
 - Questo è scorretto. Non è possibile istanziare una classe astratta come List. Dovresti utilizzare ArrayList o un'altra classe concreta.
2. ArrayList<Double> a = new ArrayList<>();
 - Questo è corretto. Si sta dichiarando una variabile a di tipo ArrayList di Double e si sta istanziando un oggetto ArrayList di Double con la sintassi del diamond operator (<>).
3. ArrayList<Double> a = new ArrayList<Number>();
 - Questo è scorretto. Non è possibile assegnare un ArrayList<Number> a una variabile di tipo ArrayList<Double>, poiché ArrayList<Number> non è un sottotipo di ArrayList<Double>. Tuttavia, puoi assegnare ArrayList<Double> a ArrayList<? extends Number>.
4. ArrayList<> a = new ArrayList<Double>();
 - Questo è scorretto. È necessario specificare il tipo al momento della dichiarazione della variabile, ad esempio ArrayList<Double> a = new ArrayList<Double>();.
5. List<Object> a = new ArrayList<Object>();
 - Questo è corretto. Puoi dichiarare una variabile di tipo List<Object> e assegnarle un oggetto ArrayList<Object>. Questo è un esempio di polimorfismo.
6. ArrayList<String> a = new ArrayList<String>();
 - Questo è corretto. Si sta dichiarando una variabile a di tipo ArrayList di String e si sta istanziando un oggetto ArrayList di String.
7. new ArrayList<()>;
 - Questo è scorretto. Devi assegnare l'oggetto creato a una variabile o utilizzarlo in qualche modo, altrimenti l'istanza verrà persa.
8. List l = new ArrayList();
 - Questo è corretto. Si sta dichiarando una variabile l di tipo List e si sta istanziando un oggetto ArrayList. È un esempio di polimorfismo, in cui si utilizza il tipo base List per riferirsi a una classe derivata ArrayList.
9. List l = new List<>();
 - Questo è scorretto. Non è possibile istanziare una classe astratta come List. Dovresti utilizzare ArrayList o un'altra classe concreta.

LA 7 È CORRETTA, PERCHÉ?

```

class E1 extends RuntimeException {}
class E2 extends E1{ }

class A {
    protected A v(int x) throws E1 { return null; }
}

class B extends A {
    @Override
    protected A v(int x) throws E1 { return null; }
}

class C extends A {
    @Override
    private A v(int x) throws E1 { return null; }
}

class D extends A {
    @Override
    public A v(int x) throws E1 { return null; }
}

class E extends A {
    @Override
    protected E v(int x) {return null;}
}

class F extends A {
    @Override
    protected A v(int x) throws E2 {return null; }
}

class G extends A {
    @Override
    protected A m(int x) throws RuntimeException { return null; }
}

```

1. Classe C:

- La classe C ridefinisce il metodo `v(int x)` dalla classe base A, il che di per sé è corretto.
- Tuttavia, il metodo ridefinito nella classe C ha un modificatore di accesso `private`, il che significa che il metodo nella classe C non può essere ereditato o accessibile da classi derivate.
- Poiché il metodo ridefinito in C è dichiarato con accesso privato, le sottoclassi come B, D, E, F, e G non possono accedervi, generando così un errore di compilazione.

2. Classe G:

- La classe G ridefinisce un metodo chiamato `m(int x)` che non esiste nella classe base A. Non c'è alcun metodo `m(int x)` con cui fare override in A.
- Inoltre, il metodo ridefinito nella classe G dichiara di lanciare una `RuntimeException`, che è una classe di eccezioni più generale rispetto a E1 (dichiarata nella classe base A). Ciò genera un errore di compilazione poiché è impossibile ridefinire un metodo con una dichiarazione di eccezione più generale rispetto a quella della classe base.

1. Classe A:

- La classe A è la classe base che definisce il metodo `v(int x)` con un modificatore di accesso `protected`. Il metodo nella classe base dichiara di lanciare E1.
- Non ci sono errori nella dichiarazione del metodo nella classe base, quindi non ci sono errori di compilazione.

2. Classe B:

- La classe B eredita dalla classe A e ridefinisce il metodo `v(int x)` in modo corretto, utilizzando il modificatore di accesso `protected` e dichiarando di lanciare E1. La ridefinizione è conforme alle regole di accesso e tipo di eccezione.
- Non ci sono errori nella ridefinizione del metodo, quindi non ci sono errori di compilazione.

3. Classe D:

- La classe D eredita dalla classe A e ridefinisce il metodo `v(int x)` in modo corretto, utilizzando il modificatore di accesso `public` e dichiarando di lanciare E1. La ridefinizione è conforme alle regole di accesso e tipo di eccezione.
- Non ci sono errori nella ridefinizione del metodo, quindi non ci sono errori di compilazione.

4. Classe E:

- La classe E eredita dalla classe A e ridefinisce il metodo `v(int x)` in modo corretto, utilizzando il modificatore di accesso `protected`. Tuttavia, il tipo di ritorno è E anziché A. Questo è accettabile in quanto E è una sottoclasse di A, quindi l'override del tipo di ritorno è valido.
- Non ci sono errori nella ridefinizione del metodo, quindi non ci sono errori di compilazione.

5. Classe F:

- La classe F eredita dalla classe A e ridefinisce il metodo `v(int x)` in modo corretto, utilizzando il modificatore di accesso `protected` e dichiarando di lanciare E2, che è una sottoclasse di E1. Questo è valido in quanto è possibile ridefinire un metodo in modo che lanci una sottoclasse di eccezione rispetto a quella dichiarata nella classe base.

- Non ci sono errori nella ridefinizione del metodo, quindi non ci sono errori di compilazione.

```

#include <iostream>
using namespace std;
class veicolo {
private:
    int pri() { return 1; }
public:
    int pub() { return 2; }

    virtual int vpub() { return 3; }
};

class camper: private veicolo {
public:
    int vpub() { return 5; }
};

class automobile: public veicolo {
private:
    int pri() { return 6; }
public:
    int vpub() { return 7; }
};
void f_automobile(automobile a) {
    cout << a.pub(); // Stampa 2 (pub() è pubblico)
    cout << a.pri(); // ERRORE: pri() è privato nella classe base
    cout << a.vpub() << endl; // Stampa 7 (vpub() è ridefinito nella classe derivata)
}

void f_p_automobile(automobile* a) {
    cout << a->pub();
    cout << a->pri();
    cout << a->vpub() << endl;
}

void f_r_automobile(automobile& a) {
    cout << a.pub();
    cout << a.pri();
    cout << a.vpub() << endl;
}

```

Nel codice fornito, la funzione `f_automobile` riceve un oggetto della classe `automobile` come argomento e tenta di chiamare i metodi `pub()`, `pri()`, e `vpub()` su di esso. Ecco cosa succede:

```

void f_automobile(automobile a) {
    cout << a.pub(); // Stampa 2 (pub() è pubblico)
    cout << a.pri(); // ERRORE: pri() è privato nella classe base
    cout << a.vpub() << endl; // Stampa 7 (vpub() è ridefinito nella classe derivata)
}

```

Quindi, la funzione `f_automobile` stamperà "2" per `a.pub()` poiché `pub()` è un metodo pubblico nella classe base. Tuttavia, genererà un errore in compilazione per `a.pri()` poiché `pri()` è un metodo privato nella classe base e non è accessibile dalla classe `automobile`.

Infine, stamperà "7" per `a.vpub()` poiché `vpub()` è un metodo virtualmente ridefinito nella classe derivata `automobile`, quindi verrà chiamata la versione della classe `automobile` del metodo.

```

class Animale{
    public Animale get(){return null;}
}
class Cane extends Animale {
    private Animale get(){return null;}
}
class Gatto extends Animale {
    public Animale get(){return null;}
}
class Topo extends Animale {
    public Gatto get(){return null;}
}

```

Nel codice fornito, il metodo `'get'` viene sovrascritto in tutte e tre le sottoclassi ('Cane', 'Gatto' e 'Topo') della classe 'Animale'. Tuttavia, ci sono errori nella compilazione relativi al metodo `'get'` nelle sottoclassi 'Cane'. Vediamo i dettagli: 1. **Il metodo 'get' di 'Cane'**: In 'Cane', il metodo `'get'` ha un livello di accesso più restrittivo rispetto a quello del metodo nella classe genitore 'Animale'. Il metodo nella classe genitore è dichiarato come `'public'`, mentre in 'Cane' è dichiarato come `'private'`. Questo è un errore nella compilazione perché il metodo sovrascritto deve avere almeno lo stesso livello di accesso del metodo nella classe genitore o un livello di accesso più ampio. Pertanto, il metodo `'get'` in 'Cane' dovrebbe essere dichiarato come `'public'` o

`protected`, ma non `private`.

Nel linguaggio Java esistono esattamente quattro livelli di accesso, sono quelli elencati di seguito partendo dal livello più “ristretto” (il meno permissivo) fino a quello più “ampio” (il più permissivo):

private (più “ristretto”)

package

protected

public (più “ampio”)

Questi elencati sono i livelli “concettuali”, ciascuno di essi rappresenta un concetto come è definito dalle specifiche del linguaggio Java. Il significato (in breve) dei livelli è il seguente:

un membro con livello di accesso private è accessibile solamente all'interno della classe in cui è definito il membro

un membro con livello di accesso package è accessibile solamente dalle classi e sotto-classi nello stesso package della classe in cui è definito il membro

un membro con livello di accesso protected è accessibile dalle classi e sotto-classi nello stesso package della classe in cui è definito il membro e in più è accessibile anche da qualunque altra sotto-classe in altri package ma solamente per effetto della ereditarietà

un membro con livello di accesso public è accessibile da qualunque classe in qualunque package

I livelli public e private sono quelli più semplici da comprendere e rappresentano in pratica i due estremi opposti, ovvero rispettivamente i concetti di: “accessibile a tutte le classi” e “accessibile a nessun'altra classe”. Invece package e protected sono due livelli intermedi che differiscono tra di loro solo per un dettaglio veramente molto “fine” e importante che verrà spiegato meglio, in maniera più approfondita, nelle sezioni successive.